Automatic Trailer Generation

Code Documentation

By: Sushant Gour- Intern/ Cloud Team- SRI-D

This document contains the explanation of each code file used in the following pipelines of the Automatic Trailer Generation Project:

(1) Impressive Speech Extraction Pipeline
(2) Horror Scenes Extraction Pipeline
(3) Trailer Stitching Pipeline

Impressive speech Extraction Part:

(1) ExtractingAudioFromVideo.py : This code is used to extract audio from a video. We need this code because we need the audio of the movie for speech transcription.

Tool used: ffmpeg

Inputs: The inputs required by this file are:

(1) video_path: It is the path where the movie (mp4) is stored
(2) video: The name of the movie (mp4)
(3) audio: The name of the audio file that will be generated by this code (mp3)

Output: An audio file (mp3) will be created in the video_path directory having the name equal to the string represented by the audio variable.

(2) SpeechExtraction1.py : This code is used for transcribing speech in a whole movie and extracting the resulting speech segments that lies inside the first x% of the movie.

This code file uses Google's Video Intelligence API for Speech Transcription. The libraries used are: io, videointelligence_v1, os, pydub, pathlib and moviepy.

Inputs: The inputs required in this file are:

(1) video_path: It is the path to the video file representing the movie (mp4)

(2) audio_path: the path to the audio file that contains the audio of the movie.

(3) output_path: the output directory where all the extracted speech segments will get stored.

(4) x: it is a number between 0 and 100 which represents the percentage of speech from the start that we are considering. This threshold is used to avoid spoilers as spoilers are generally present towards the end of the movie. The ideal value for x is 80.

(5) language_code: It is the language code of the language in the movie. The language codes can be found from GCP platform. For American English, the code is: en-US.

Output: The output of this file is all the speech segments which lies inside the first x% of the speeches in the movie. The last (100-x)% speeches are not extracted as they might contains spoilers. The output format of the speech segments is "wav". The ideal value for x is 80.

The output file names itself contains the timestamps of the corresponding speech segments in the movie in seconds. The output file names have the following format: (startTime,endTime).wav , where startTime and endTime are in seconds. The output files gets stored in the output_path directory.

(3) SpeechExtraction2.py : This file calculates the outlier scores of all the speech segments which were extracted from the movie and sorts the segments in descending order of their outlier scores and choose as much segments from the top that fits inside the duration of speech we need in the trailer. After that for the above chosen most impressive speech segments it extracts the corresponding scenes from the original movie and stores them in the output_folder.

The outlier scores are calculated by using Gaussian Mixture Model which is a clustering model. We cluster all the speech segments based on the following features: Chroma Vector, MFCC's and Delta MFCC's. The most impressive speech segments will obviously have different features than the normal speeches and hence they will be present in separate clusters. Our model assigns high outlier scores to speech segments present in such clusters. Hence, finally, the outlier score represents how impressive a particular speech segment is. More the outlier score, more is the impressiveness.

The libraries used are: io, os, re, librosa, numpy, matplotlib, math, sklearn and moviepy.

Inputs: The inputs to this file are:

(1) path : path to the folder where all the speech segments which were extracted from the movie using SpeechExtraction1.py are stored.

(2) movie_path : the path to the movie file (mp4) from which we will extract the scenes corresponding to the most impressive speeches.

(3) output_folder : the path of the folder where all the extracted impressive speeches will get stored.

(4) totalDurationOfSpeech : it is the maximum duration of speeches (in seconds) which we can add in our trailer. Actually, it signifies the percentage of speech in the trailer.

(5) lowerThreshold : it is the lower limit on the length of an output speech segment to be able to get included in our trailer

(6) upperThreshold : it is the upper limit on the length of an output speech segment to be able to get included in our trailer

We are putting the lowerThreshold because a speech of say 1 second duration will be too small and will look weird in our trailer.

We are putting the upperThreshold because a speech of say 20 second duration will be too large and will make our trailer boring.

Output: The output of this code is the most impressive speech segments (mp4) extracted from the movie which fits inside the duration of speech we want in our trailer. All the files (mp4) gets stored in the output_folder. The output files have names in the format: (startTime,endTime) ,where startTime and endTime are in seconds. Now, let us suppose that we want a maximum of 60 seconds of speech in our trailer, then the output of this code file is as much most impressive speech segments from the movie such that their sum is less than or equal to 60 seconds. The speeches are extracted in descending order of their outlier scores. Also, we can also decide the thresholds on the length of the speech segments. In my code, I have put the threshold such that the output speech segments are having length in the range: [3,10] seconds, i.e., I have taken only those speech segments which have a length greater than or equal to 3 seconds and less than or equal to 10 seconds as a speech segment of length< 3 sec wouldn't have much speech in it and a segment of length> 10 sec would be too large and will make the trailer look boring. We can adjust the threshold in the speech according to our need. In conclusion, this code file outputs the impressive speech segments in descending order of their outlier scores and takes only those speeches which have length in the range: [lowerThreshold,upperThreshold]. Also, the total duration of the extracted speeches doesn't exceeds the "totalDuration" of speech we want in our final trailer.

Horror Scenes Extraction Part:

(4) mp3Towav.py : This code is used to convert an mp3 file to a wav file. We use this code because the input audio used in the horror scenes extraction pipeline has to be in wav format.

Tool used: ffmpeg

Inputs: The inputs required by this file are:

(1) audio_path: It is the path where the audio of the movie (mp3) is stored
(2) audio_mp3: The name of the audio file of the movie (mp3)
(3) audio_wav: The name of the audio file that will be generated by this code (wav)

Output: An audio file (wav) will be created in the audio_path directory having the name equal to the string represented by the audio_wav variable.

(5) HorrorScenesExtraction.py : This code file is used to extract horror scenes from a movie.

The libraries used are: librosa, moviepy, os and pandas.

The inputs and outputs of this file are given below:

Inputs: The file requires the following inputs:

(1) audio_path: This variable stores the path to the audio (in wav format) of the movie. Note that this audio file should be of wav format.
(2) totalDurationOfHorrorScenes: This variable represents the total duration of horror scenes we want in our trailer.
(3) movie_path: This variable stores the path of the original movie (mp4) from which the horror scenes are to be extracted.
(4) output_folder: This variable stores the path of the output folder in which the extracted horror scenes are going to get stored.

Output: The code extracts horror scenes from the movie and stores them in the output_path folder. All the horror scenes are having mp4 format. Also, the sum of all the horror scenes does not exceeds the total duration of horror scenes that we want in our trailer. The output files are having the names as: (startTime,endTime) where startTime is the starting time of that horror scene in the movie and endTime is the ending time of that horror scene in the movie.

How we extract the horror scenes:

We run a 5 second window from start to end of the audio file of the movie and calculate short-time energy of each window. We sort the windows in descending order of their energies. More the energy of a window, more loud is the audio in that window and hence more probability that the scenes in the window is a horror scene. As the windows are sorted in descending order of their loudness, we extract as much windows in decreasing order of their loudness such the sum of all extracted windows doesn't exceed the total duration of horror scenes that we want in our trailer. Note that each window is named according to the format: (startTime,endTime) ,where startTime and endTime are in seconds.

Trailer Stitching Part:

Now, I assume that I have the Speech scenes, the Action/Horror scenes, the Wide scenes, the Title scenes in one folder and the Character scenes in a separate folder, i.e. all the scenes are together in one folder except the Character scenes. The character scenes are stored in a separate folder. This is because we will add a smaller duration of fade in and fade out in the character scenes as compared to all other scenes. So the character scenes should be separately stored.

(6) RemoveClashes.py : We can have cases when a number of scenes are overlapping. This code removes this type of clashes. It finds the union of all the scenes that are having overlap and it discards all those scenes and extracts their union from the movie. For example say three scenes have overlap: (10,15) , (12,17) and (15,20) , then this code discards all these scenes and extracts their union i.e. (10,20) from the movie. The clashes can be observed directly through the names of the files as the file names are in the format: (startTime,endTime) , where startTime and endTime are in seconds.

Libraries used: os, re and moviepy.

Inputs: The inputs required in this file are:

(1) source: The path to the source folder where all the scenes are located.
(2) destination: The path to the destination folder where the output is going to get stored.
(3) movie_path: The path to the movie (mp4).

Output: This code traverse in the source folder and check for overlaps in the files. If there is overlap in some files, then it discards them and extract their union from the movie and store it in the destination folder. The files that don't have clashes with other files are moved directly to the destination folder. Eventually, the files in the destination folder are free from clashes.

Note: As we have two folders with us: one having all the scenes except the character scenes and the other having only the character scenes. We will run the RemoveClashes.py code only on the first folder as there is no need to check the character scenes for clashes as the character scenes are only facial clips of characters which are of barely 1 to 2 seconds duration. So they won't clash with any other scene in the trailer.

(7) FadeInFadeOut.py : This code is to apply fade in and fade out effect to all the scenes that are to be stitched together. I assume that we have all the scenes on which we want to apply fade in and fade out effect in a folder.

Tool used: ffmpeg

The inputs and outputs of this file are given below:

Inputs: This file requires the following inputs:

(1) path: This variable stores the path to the folder that contains all those scenes on which we want to add fade in and fade out effect.
(2) fadeDuration: This is the fade duration in seconds which we want to apply. A fade in effect of length = fadeDuration and a fade out effect of length = fadeDuration will be applied to each scene. The fade in and fade out effects are applied to both the video and the audio of a particular scene.

Output: For each file in the "path" folder, a new file is created which is having fade in and fade out effect in it. The name of the new file is = oldFileName + "_WithFadeIn".

Now, at the end of the code, we delete all the old files from the "path" directory and hence, only the new files which have fade in and fade out effects in them are left in the "path" directory.

Note: For better results, run the FadeInFadeOut.py file on character scenes separately with a fade duration of 0.2 seconds as the character scenes are of about 1-2 seconds. Hence, we should apply a lesser fade duration (0.2 sec) as compared to all other scenes (0.5 sec). So, in conclusion, as we have the character scenes in one folder and rest of the scenes in other folder (which are without clashes), so we use fade in fade out code on both folders separately with different fade durations. At the end merge the contents of both folders using the below code file:

(8) MergeContentsOfTwoFolders.py: This code is used to merge contents of two folders. As we are applying fade in and fade out on character scenes separately, so we have to merge the contents of both the folders at the end. One folder contains character scenes with fade effect applied to them (fade duration = 0.2 sec) and other folder contains rest of the scenes with fade effect applied to them(fade duration = 0.5 sec).

Libraries used: os

Inputs: The inputs required by this file are:

(1) path1: path to the folder containing character shots with fade effect
(2) path2: path to the folder containing rest of the scenes with fade effect

(3) merged_path: path to the folder where we want to store the contents of path1 and path2

Output: The contents of both the folders path1 and path2 will be moved to the merged_path folder.

Now in the merged_path folder, we have all the scenes we want in our trailer with fade in and fade out effects in them.

(9) IncreaseVolumeOfScenesThatContainsSpeech.py:

Generally, the background music is louder than the speech in the trailer. So it may happen that the speech gets suppressed by the BGM. So, we use this code to increase the volume of the scenes containing speech in them by a factor of 4. The factor can be varied accordingly. Note that the action scenes and wide scenes can also contain speech in them. We assume that we have all the scenes in a source folder. This code uses Speech Transcription to tell whether a scene has even a little bit of speech in it or not. If it has speech, then we move that scene to an intermediate folder. Now, all the scenes having speech in them are moved to the intermediate folder and only the scenes which don't have speech in them are left in the source folder. After that we increase volume of all the files in the intermediate folder. After that, we move the files in the intermediate folder back to the source folder. Finally, the source folder contains all the scenes we want in our trailer with an increase in volume by 4x in the scenes having speech in them. I have used a volume factor of 4x as it was giving me the best results.

Tool used: ffmpeg

Inputs: The inputs required in this file are:

(1) source_path: It is the path to the source folder which contains all the scenes.
(2) intermediate_path: It is the path to the intermediate folder which will temporarily store the scenes having speech in them.
(3) language_code: It is the language code of the language in the movie. The language codes can be found from GCP platform. For American English, the code is: en-US.
(4) volume: It is the factor by which the volume is to be increased.

Output: This code moves the files having speech in it to the intermediate folder, increases their volume and again moves the files back to the source folder. Ultimately, the source folder have all the files with the volume of the files having speech in them increased by a factor equal to the value inside the "volume" variable.

After the execution of the code, the files having speech in them will have their name in the format: (startTime,endTime)_Amplified , where startTime and endTime are in seconds. The files which don't have speech in them will have names in the format: (startTime,endTime) , where startTime and endTime are in seconds.

(10) Concatenate.py:

Now, as we have all the scenes with us having the necessary modifications such as fade in fade out effects and increased volume of scenes having speech, we are now ready to concatenate all the scenes and make the trailer. This code concatenates all the scenes in chronological order of their timestamps. The value of timestamps can be easily extracted from the names of the files itself as the files are having names in the format: (startTime,endTime). The code stores all the filenames in a list and sorts the list according to increasing order of their startTime and concatenates the files in that order. So, we will get the trailer as output and the scenes in the trailer are in the same order as they appear in the original movie.

Our code works in the following way: Firstly, it converts all the mp4 files in the source directory into (.ts) format. We do this because (.ts) is a lossless format and hence while concatenating ts files, we don’t get any distortions in video and audio at the boundary of two files. Now after that our code makes a text file which contains names of the ts files line by line in chronological order. Now, we traverse line by line in the text file and concatenates the corresponding ts files. At the end, we convert the final ts file back to mp4 format. Hence, we get our trailer in mp4 format using this code. The trailer gets stored in the source_path directory itself.

Libraries used: os, ffmeg, moviepy and re

Inputs: This file requires the following inputs:

(1) source_path: path to the folder where all the scenes to be concatenated are stored.
(2) trailerName: It is the name that we want to assign to our trailer.

Output: The output of this code is an mp4 file which is formed by the concatenation of all the files in the source folder in chronological order. The output file is created in the source_path directory itself and is having name equal to the string stored in the trailerName variable.

(11) MoveFileFromOneFolderToAnother.py:

Now, to apply Background music to the trailer, we need to make sure that the theme music file and the trailer file are in the same directory. Hence, we use this file to move the theme music and the trailer file to the same directory.

Libraries used: os

Inputs: The inputs to this file are:

(1) source_path: It is the path to the directory in which the file to be moved is present.
(2) destination_path: It is the destination directory where we want to move the input file.
(3) filename: It is the name of the file to be moved (mp4)

Output: The input file is moved from source_path to destination_path.

Use this code to move the theme music file (mp3) and the trailer file (mp4) to the same directory.

(12) AddBackgroundMusic.py:

Now the final step is to add the theme music in the background of the trailer. We assume that we have the Theme Music file (mp3) and the Trailer file (mp4) in the same directory. This code makes a new mp4 file which has the theme music in the background of the trailer. The theme music is overlayed on the music already present in the trailer. The duration of the final output file will be equal to the minimum value among the duration of the trailer and the duration of the theme music. We assume that the theme music duration is always greater than the duration of the trailer so that the output file will have the same duration as that of the trailer.

Libraries used: ffmpeg and os

Inputs: The inputs required by this file are:

(1) source_path: It is the path to the directory where the theme music and trailer is located.
(2) theme_music: It is the name of the theme music file (mp3).
(3) input_video_name: It is the name of the trailer file (without BGM).
(4) output_file_name: It is the name that we want to give to our final trailer with background music.

Output: The final trailer is generated (mp4) that contains theme music in background. The final trailer is created in the source_path directory itself and has the name equal to the string stored in the output_file_name variable.