– Independent Work Report Fall, 2016 –

Texty: A Tool for Easy Video Subtitles and Benchmarking Framework for Speech-to-text APIs

Ragy Morkos

Adviser: Professor Alan Kaplan

Abstract

The trend of viewing videos via smartphones and tablets rather than through computers has been continuously rising. However, the proliferation of video content usage on tablets and smartphones leaves out many individuals who suffer from a form of hearing impairment as well as language learners who are not fluent in the video's language.

In this paper, we explore the possibility of an Android application that uses off-the-shelf speech recognition technologies to produce an SRT (subtitle file) that can show subtitles during video playback using almost any well-known Android media player. The bulk of this paper is dedicated to the subtitle alignment algorithm that is proposed, since off-the-shelf speech recognition technologies only provide a mere transcript without any timecodes.

Moreover, this application can simultaneously offer a system that will exploit real user feedback regarding the accuracy of the transcription and subtitle timing. This feedback can be valuable for the speech recognition APIs, as well as to further enhance and tweak the subtitle alignment algorithm. Our experimental results show great potential for this Android application, providing completely automatic subtitles with decent quality and performing better than the currently available automatic subtitle generation alternatives.

1. Introduction

One of the most valuable means of information and communication on the internet comes from video [1, 2], and, increasingly, from video viewed on mobile devices and tablets [3]. In fact, nearly half (48%) of millennials only watch videos on their mobile devices, and they are three times more likely than baby boomers to watch a video on their mobile device.[1] This trend is only growing with each year; as a simple example, the share of mobile video plays has gone up an incredible 2,048% over the past five years, and at the end of 2015, mobile video plays (smartphones and tablets) comprised almost half (46%) of all video plays.[2] This trend is expected to increase further with the proliferation of 4G LTE networks and the ever growing ubiquity of Wi-Fi.

While the proliferation of mobile video as a medium of communication and information is enormously beneficial, it currently leaves out people who cannot make use of it due to hearing impairments or language disabilities. It is estimated that over 5% (360 million) of the world's population has some form of disabling hearing loss, with 48 million in the United States alone.[3] Inability to fully understand the video's language can also provide equivalent difficulties in making use of the medium of video. In fact, a study in the Journal of Learning Disabilities compared difficulties exhibited by students learning a second language to those with learning disabilities [4]. With much of the online video content being in different languages without subtitles, language learners do not have equal access to this video material. Videos sent between individuals on messaging apps, for instance, have no way to be understood if they are in a language the recipient is not fluent in. Sometimes understanding information in another language

[1] https://animoto.com/blog/business/millennials-video-infographic/
[2] http://go.ooyala.com/rs/447-EQK-225/images/Ooyala-Global-Video-Index-Q4-2015.pdf
[3] http://www.who.int/mediacentre/factsheets/fs300/en/

can prove to be crucial, as in the case of the refugee crisis today [5]. So much that is the case that Global Citizen characterized the difficulty in speaking and learning English as one of the "biggest challenges facing refugees and immigrants in the US."[4] Interestingly, smart phones are incredibly beneficial to refugees, since its mobility and communication usefulness makes it for one of the most important belongings to communicate important information and to contact family back home as well as for GPS on their journeys. Researchers according to Pennsylvania State University showed that in a Syrian refugee camp, "86% of youth own a mobile handset and more than 50% use the internet at least once a day".[5]

Automatic subtitles can increase the accessibility of user-generated video, in particular on mobile platforms. For people with hearing impairments, it will deliver for them the much-needed audible information in the video in a textual form that can make any video with speech accessible to them. For language learners, the learning theory literature has shown that the correlation between close captions or subtitles and better comprehension of the language is not a small one, and that subtitles can indeed offer the much needed textual representation of speech that language learners can easily understand [6, 7, 8]. In the case of refugees messaging news video content to each other, having a way to automatically have the video subtitled is incredibly valuable. Despite all of these potential benefits that can reach millions who need them, there does not exist a technology that can provide automatic subtitles. Off-the-shelf speech to text APIs and services are abundant and provide relatively accurate transcripts of video or audio files. Unfortunately, they only provide transcripts of the speech or audio, and not the timecoded transcripts needed for subtitles.

[4] https://www.globalcitizen.org/en/content/the-7-biggest-challenges-facing-refugees-and-immig/
[5] http://news.psu.edu/story/350156/2015/03/26/research/ist-researchers-explore-technology-use-syrian-refugee-camp

This paper focuses on the goal of providing a simple, yet effective Android application that can provide automatic subtitles using only off-the-shelf speech recognition APIs. This application will be incredibly helpful to individuals with hearing impairments or to language learners. As noted before, more and more videos are viewed on smartphones and tablets, and hence our decision to develop this software as a mobile application to reach as many people as possible. We chose the Android mobile operating system because it has the largest user base among all mobile operating systems, with about 87.5% of the global smartphone market (9 in every 10 smartphones) on Android.[6]

Our paper is organized as follows. In Section 2, we discuss related work and background, most notably previous work and algorithms that have tried to use off-the-shelf speech-to-text APIs to produce reasonably accurate timecoded transcripts (subtitles), as well as briefly look at YouTube's automatic closed-captioning system. In Section 3, we discuss our approach to developing our application, most importantly our unique subtitle alignment algorithm as well as our experimental design and how we will test it. We will also discuss our development of our benchmarking technique for the speech-to-text APIs, as well as for fine-tuning our subtitle alignment algorithm. In Section 4, we provide detailed description of our subtitle generation application. Details of our benchmarking implementation and our feedback functionality to fine-tune our subtitle alignment algorithm over time are also presented. In Section 5, we provide our evaluation results of the Android application, and how it compares with YouTube's automatic closed-captioning functionality as well as other possible subtitle alignment algorithms. In Section 6, we discuss future work. Finally, we conclude in Section 7.

[6] http://www.cnbc.com/2016/11/03/google-android-hits-market-share-record-with-nearly-9-in-every-10-smartphones-using-it.html

2. Background and Related Work

2.1 Historical Background

Subtitles have been around since the first forms of media started appearing in our world. Before the creation of image streams that could simultaneously play an audio stream (i.e., videos we know today), videos were simply a bunch of images quickly and simultaneously being projected on the screen. Because there was no audio stream, viewers could not hear speech of the characters, and thus text was viewed on the cards in order to show what the characters were saying. This was the birth of subtitles as we know it today [9].

Today, subtitles are to be found in films, TV series, DVDs, and Blu-rays. These subtitles are all developed "off-line" by professional humans in order to provide accurate timecoded transcripts ready for viewing. However, the trend for "live" closed-captioning or subtitles is growing. For instance, the Federal Communications Commission in the United States mentions that "Congress requires video programming distributors (VPDs) - cable operators, broadcasters, satellite distributors and other multi-channel video programming distributors - to close caption their TV programs."[7] It is worth noting, however, that this is not the case for video material on the internet, unless it was previously broadcasted on TV.[8]

Most of the internet's video content, therefore, does not have subtitles, and so user-generated videos downloaded from the internet or videos exchanged between individuals on messaging platforms do not come with subtitle options. The reason why subtitles are not more widely available is due to the fact that they are expensive and extremely time-consuming to

[7] https://www.fcc.gov/consumers/guides/closed-captioning-television
[8] https://www.fcc.gov/consumers/guides/captioning-internet-video-programming

produce. Currently, most closed-captions are produced by professional software that must still be operated by a human, and so the process is far from automated.

2.2 YouTube's closed-captioning functionality

Currently, the only service that provides completely automated closed-captions is YouTube. The automatic captioning functionality was announced on the Google blog in 2009, and has since then become available on most of YouTube's videos.[9] The functionality utilizes Google's automatic speech recognition (ASR) technology to provide timecoded transcripts. Although free, the quality of YouTube's closed-captions is, for the most part, poor. A BBC article in 2015 discussed this decrease in quality, which featured one famous YouTube vlogger who was deaf, urging the YouTube community to manually close caption videos since YouTube's automatic captions quality was so bad.[10] Despite YouTube's close captions being less than ideal, they are currently the only fully automated service for providing close captions. Details about the caption alignment algorithm behind the service, however, has never been made public. Nevertheless, in our evaluation of our Android Application, we compare and contrast our subtitle quality against those produced by YouTube's system.

2.3 Using Audio Markups for Caption Alignment

Using audio markups for caption alignment is an idea proposed by [10]. An audio markup is simply a spoken sequence of words that can be transcribed by the ASR, but at the same time does not naturally occur in normal speech. An example of an audio markup can be the sequence of letters "AAA". Their proposed mechanism for automatic generation of subtitles is

[9] https://googleblog.blogspot.com/2009/11/automatic-captions-in-youtube.html
[10] http://www.bbc.co.uk/newsbeat/article/31004497/youtube-we-know-automatic-subtitles-arent-good-enough

based on off-the-shelf ASR APIs. The algorithm processes the whole audio stream, looking for silence frames in speech (based on sound energy level). Once it detects a certain number of silence frames, it inserts an audio markup in the middle of this silence frame and saves the time at which it occurs, and continues to do so for every silence frame. Next, the audio stream with the audio markups gets passed onto an ASR API, and the output is a transcription of the original audio stream with the transcribed audio markups in all the moments of silence. The audio markup transcriptions in the transcribed text provide an anchor for the timing codes, and the timecoded subtitles are generated that way with the transcribed audio markups getting removed from the final output file.

It is worth noting that the quality of the produced subtitles in their experiment mainly depended upon two variables: the first is the threshold for silence used and the second is the distance from the last inserted markup. Trying out different values for these two variables gave different results. This is mostly due to the fact that silence lengths and their recurrence occur in many different permutations according to the speaker's speech patterns. It is also worth noting that the presence of audio markups in many cases interfered with the ASR API's accuracy, though not by much [10].

3. Approach and Design

3.1 Approach to Subtitle Alignment

Our approach to the subtitle alignment mechanism is slightly different than the approach outlined in [10]. The basic idea of our mechanism is that instead of inserting audio markups in the middle of silence frames, we instead opted to chunk all speech segments between silence frames, pass each resultant small audio file with speech into the ASR API, and obtain a transcription of that part of speech. Since we have already known where each silence period is at, we can timecode the resultant transcription accordingly.

One important advantage over [10]'s proposed mechanism is that our algorithm does not alter the original audio stream by inserting audio markups. In Section 5, we compare results from [10]'s implemented algorithm with our proposed approach, and we indeed notice that our approach results in slightly more accurate transcription results. Nevertheless, the "chunking" of speech segments and passing each chunk onto the ASR API is time consuming, with the whole process taking 25% of the video's total running time.

3.2 Approach to Benchmarking

Because there are various existing speech-to-text APIs, choosing just one of them can severely limit our proposal. This is due to several reasons, most importantly the fact that an API can be discontinued from its provider, and thus our Android application would be no longer functional. To make sure this does not happen, our framework is modularized, allowing any ASR API to be used.

Moreover, since we use a unique chunking approach for obtaining the speech segments, it is possible that it can interfere with ASR APIs that utilize speech context information to increase recognition accuracy. Therefore, we have implemented a benchmarking module that works on user feedback. The module's implementation is as follows. When a user opens our application and passes into it a video from his phone's memory to have it subtitled, our code randomly chooses an ASR API from all the ones defined in the program. The program then uses the ASR API to produce the subtitles using the mechanism described above in Section 3.1, and the subtitles are produced as an SRT file that will show the subtitles when the video is played again using almost any Android video player. The next time the user opens the app, they are prompted a feedback form to fill out. This feedback asks the user about the quality of the subtitles (i.e., transcription accuracy) as well as its timing accuracy. Over time, the feedback concerning the quality of the subtitles will shed light on which ASR API offers the best results with our "chunking" mechanism of speech segments. It can also provide feedback that can be useful to the ASR engines, further enhancing their quality. Finally, the feedback regarding timing accuracy of the subtitles can be very useful for further enhancing the subtitle generation algorithm. By getting real user data, we would be able to tweak some of the algorithm's important variables, most notably the silence percentage threshold, the silence period definition (i.e., minimum number of consecutive silence frames for this group of consecutive silent frames to be considered a "silent period"), and, finally, the "distance from the last silence period" threshold that defines what is the minimum amount of non-silent frames that have to exist before another silence period can exist. These three variables will be discussed in more detail in Section 4.1. In summary, having this kind of feedback information can be helpful for honing the algorithm to different types of speech (for example, a news broadcast, regular speech, advertisement, etc.). Once this information is garnered, statistical analysis of the dataset could be implemented, and perhaps different threshold variables can be used for different types of speech and/or videos.

4. Implementation Details

4.1 Implementation Details of Subtitle Alignment

This section will address our proposed mechanism for subtitle alignment. Figure 1 shows the architecture of our mechanism.

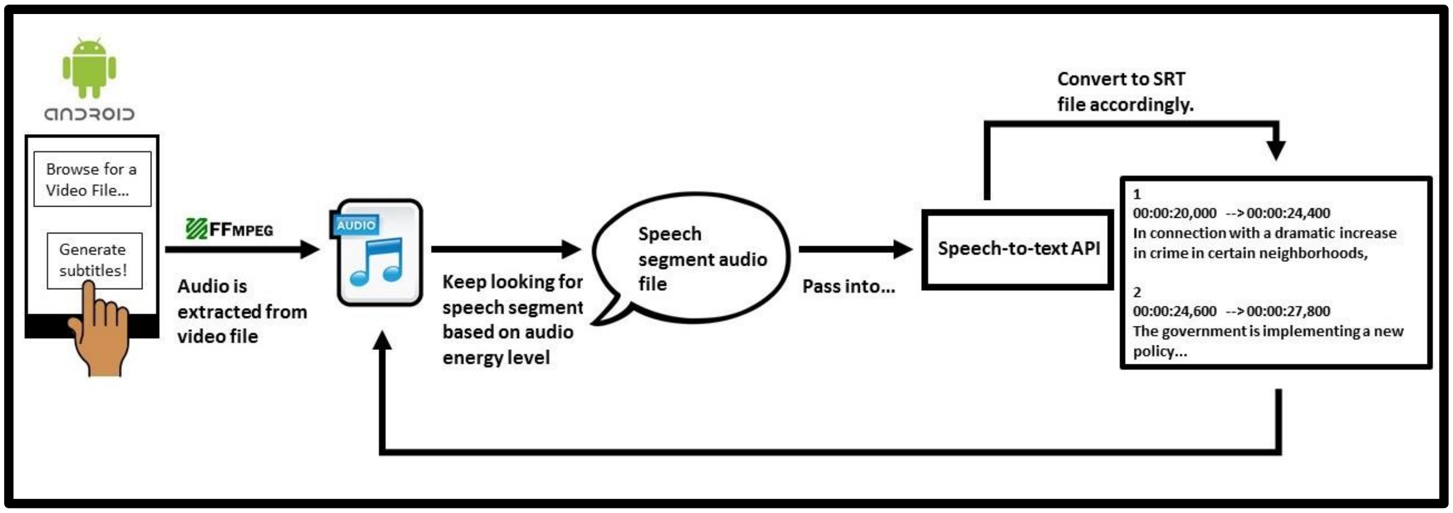

ANDROID
Browse for a Video File...
Generate subtitles!
FFMPEG
Audio is extracted from video file
AUDIO
Keep looking for speech segment based on audio energy level
Speech segment audio file
Pass into...
Speech-to-text API
Convert to SRT file accordingly.
1
00:00:20,000 --> 00:00:24,400
In connection with a dramatic increase in crime in certain neighborhoods,
2
00:00:24,600 --> 00:00:27,800
The government is implementing a new policy...

Figure 1: Implementation Details Overview

The implementation details are as follows. The user first browses from the app a video file in their phone's memory, then presses a button to start generating the subtitles. This starts the subtitle generation process. First, the audio portion of the video file is extracted, using the FFmpeg open source library. Next, the program assesses the number of audio samples per second that composes the audio stream. We build upon the work of [10] (described in Section 2.3), and define the audio stream as a "sequence of *N virtual audio frames*, $A = a_1, a_2, \ldots, a_N$, where the time length of each audio frame is fixed and equal to 30ms (a common value when analyzing an audio signal)" [10]. Knowing the number of audio samples per second is important to know how many samples to process within each audio frame (i.e. within the 30ms intervals suggested above).

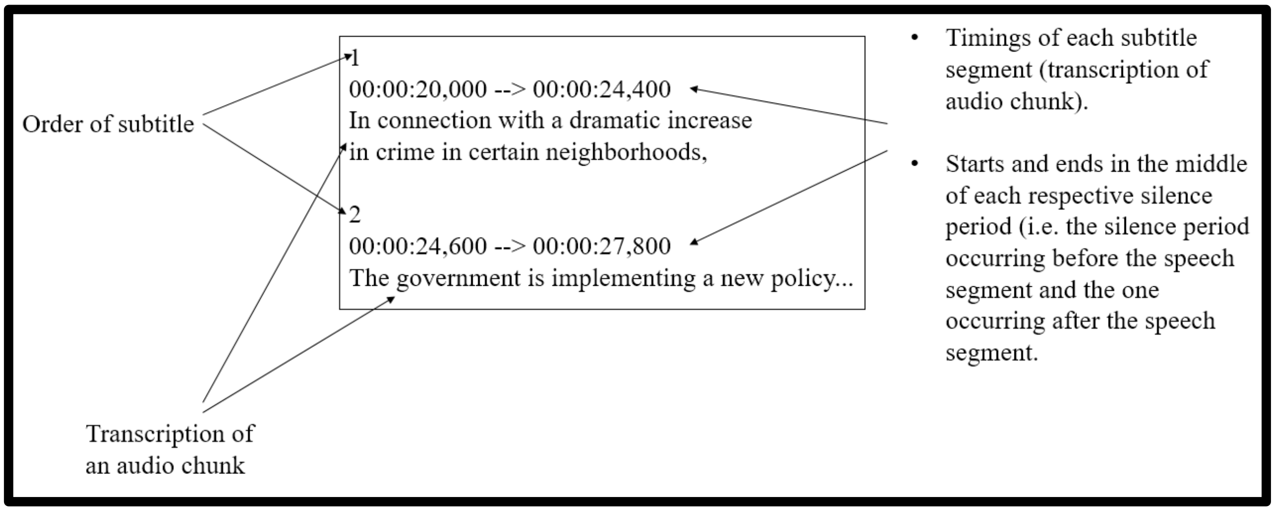
After the number of audio frames is calculated, our mechanism, similar to [10], analyzes each frame to find silence periods. [10] propose a silence period to be calculated based upon

$$Energy_i = \frac{\sqrt{\sum_{n=1}^{N} pcm_n^2}}{N}$$

sound energy. The formula used in their implementation, and in ours, can be written as follows: "where N is the number of audio samples within the frame and pcm_i is the *i-th* audio sample of the considered frame" [10]. This value is then compared to the sound energy silence threshold, and, if below it, this audio frame would be considered as a silent frame.

Because every video passed onto the application might have different and *changing* acoustic properties (i.e. individuals speak with different average sound levels, background noise can range from nonexistent to present, sound in video can vary in energy level, etc.), the silence threshold is dynamically calculated each time a different video is passed into the application. [11] proposed this dynamic readjustment by which the sound energies are calculated for every frame in the audio stream, and the silence threshold is calculated as a certain lower percentage of all the audio energy levels (for instance, the value that borders the lowest 20%). While this process is a bit computationally intensive, especially for lengthy videos, it was proven to be much more effective in our evaluation as opposed to using a fixed sound energy constant for every video. Furthermore, this calculation technique was used in [10]'s proposal and was also reported to be effective [10]. It is worth noting that [10] propose tweaking this silence threshold percentage with different types of video (i.e. user generated video capture, a news segment, etc.), something we will address in Section 4.2 of this paper.

After the calculation of the number of audio samples within each audio frame and calculating the silence threshold, our "chunking" mechanism can now start on the audio stream. We first create and open an empty SRT (subtitle) file with the ".srt" extension for the timecoded transcripts to be written. Next, for every audio frame, we calculate whether its audio energy level falls above or below the silence threshold. If it does, it indicates that there exists speech in this frame. The next frame's audio energy level is calculated, and so on until we reach a silence frame. If the number of consecutive silence frames reaches the value of the silence period constant, and if the distance (number of frames) to the last silence period is greater than the constant defining the "distance from the last silence period", we can consider this a viable chunk of speech that we extract using the FFmpeg library and pass it into the ASR API. The transcribed text obtained from the API would then be written to the ".srt" file, with the time codes defined as following in Figure 2.



```
1
00:00:20,000 --> 00:00:24,400
In connection with a dramatic increase
in crime in certain neighborhoods,

2
00:00:24,600 --> 00:00:27,800
The government is implementing a new policy...
```

Order of subtitle
Transcription of an audio chunk
• Timings of each subtitle segment (transcription of audio chunk).
• Starts and ends in the middle of each respective silence period (i.e. the silence period occurring before the speech segment and the one occurring after the speech segment.

Figure 2: Timing Transcripts of Audio Chunks based on Location of Silence Periods

It is worth noting that, while there might be audio chunks without speech (i.e. high sound energy due to background noise), these audio chunks, when passed into the ASR API, might not produce any actual text since it would not be recognized as speech. Furthermore, almost all ASR APIs return information about the accuracy of their transcriptions. In other words, we can always detect in our mechanism whether the transcription could have been for incoherent white noise speech based upon its accuracy level. If the accuracy level was too low, we can always disregard this audio “chunk”, since including it might simply produce faulty transcriptions. While we did not face this problem when experimenting with videos in our evaluation, it is nonetheless a benign assertion that could prevent faulty transcriptions from making their way to the end subtitle file.

For the purposes of completion, we have included pseudo code of our mechanism (Appendix A). We have decided not to include much of the specific Java jargon so as to present our mechanism as abstractly as possible; that way, our pseudo code should provide a basis for our application to be implemented on any platform. Moreover, some of this code can be made more efficient; however, for purposes of demonstration and clarity, we tried to be as explicit as possible when writing it.

4.2 Implementation Details of Benchmarking Framework

In order to collect user feedback and data regarding our mechanism, we have created a Firebase database that can hold all the user feedback. In Figure 3, we show an overview of our architecture.

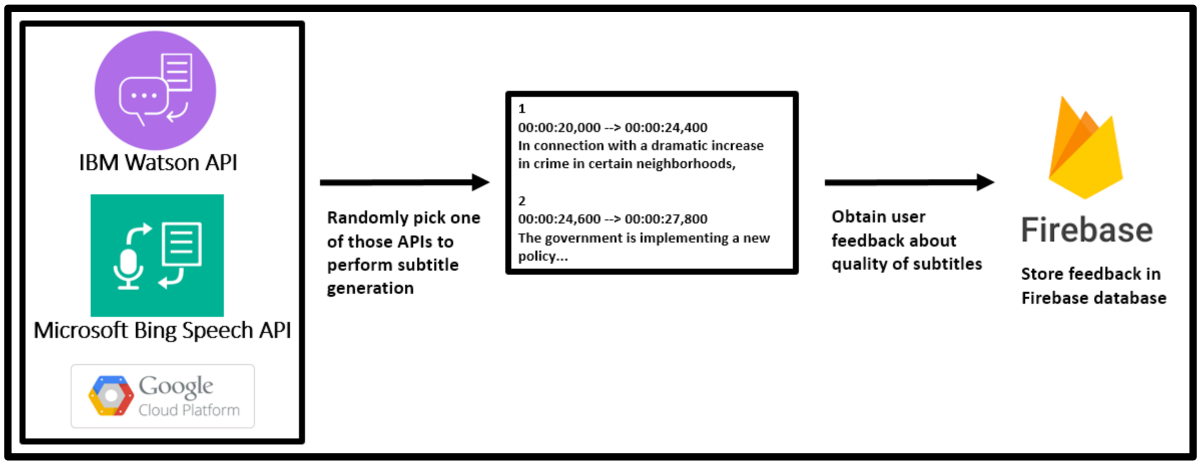
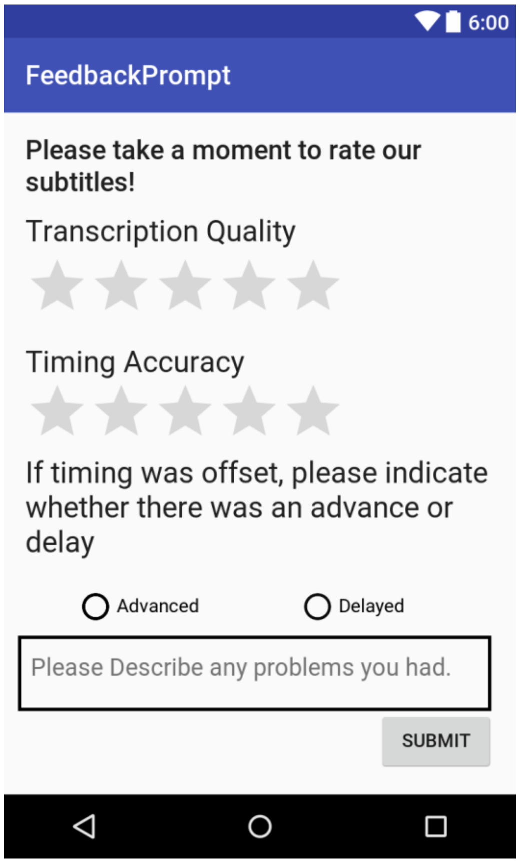

IBM Watson API
Microsoft Bing Speech API
Google Cloud Platform
Randomly pick one of those APIs to perform subtitle generation
1
00:00:20,000 --> 00:00:24,400
In connection with a dramatic increase in crime in certain neighborhoods,
2
00:00:24,600 --> 00:00:27,800
The government is implementing a new policy...
Obtain user feedback about quality of subtitles
Firebase
Store feedback in Firebase database

Figure 3: Implementation Overview of Benchmarking Framework

The benchmarking framework can be described as follows. When a user first chooses a video file to subtitle using the application, one ASR API is randomly selected among all the APIs configured in the application (in Figure 3, we only show three of those APIs, namely Google Speech, Microsoft Bing's, and the IBM Watson API, but since our code is modularized, adding extra APIs should not require more overhead). Next, this API is used for the transcription process of the video's speech as described in Section 4.1. After the subtitles are generated and the user utilizes them when watching the video, when they next open the application, they will be prompted a feedback form asking them questions regarding transcription quality and timing accuracy. Figure 4 shows a prototype for such a feedback form on our Android application.



Please take a moment to rate our subtitles!

Transcription Quality

Timing Accuracy

If timing was offset, please indicate whether there was an advance or delay

Advanced Delayed

Please Describe any problems you had.

Figure 4: Prototype for Feedback prompt

The datum obtained from the user is stored accordingly in the database holding that specific API's data. Over time, this information, alongside with any written user feedback, will enable us to assess which ASR API works best with our speech segments chunking mechanism (i.e. which API(s) do not have their word error rate (WOE) increasing due to our chunking mechanism). Furthermore, as mentioned in Section 3.2, this feedback is extremely important in order to tweak the different variables of the algorithm, namely the silence percentage threshold, the silence period definition, and the "distance from the last silence period" threshold.

5. Evaluation Results

For our evaluation, we tested our application on fifteen different video files, ranging in length from twenty seconds to three minutes, and ranging in content from user-generated content similar to the one exchanged via video messaging apps as well as short news segments where there is more than one speaker. In order to produce uniformity in our results, we tested each of our ASR APIs on each of our videos. Since our benchmarking framework would need a large pool of user feedback in order to start producing statistically significant results, it obviously would not have been feasible, nor useful, to randomly select APIs as we proposed in Section 5.2 for the evaluation of the speech segments chunking mechanism.

We compared our test results with both YouTube's close captioning mechanism and also an implementation of [10]'s caption alignment mechanism using audio markup insertion. It is worth noting that YouTube does not currently offer a close captioning functionality for videos not hosted on its platform. One roundabout users exploit is to upload the video privately on their account, then extract the close captions as an SRT file and take the video down afterwards. This is, of course, tedious to users to say the least. However, we do utilize this roundabout functionality in order to obtain YouTube's close captioning results for the same videos we used when assessing our algorithm.

5.1 Performance against the Markup Insertion Algorithm

We will first compare our implementation results to those of [10]'s audio markup insertion mechanism for the same videos, and using the same ASR API for each subtitle generation procedure, averaging over the results of all APIs. Because both of our implementations have similar constants utilized in the respective algorithm, we first needed to assess the constant values that worked best for both of our algorithms. The two most important constants are the silence percentage threshold and the minimum distance to the last silence segment. For both of those variables, our experimentation results have shown that setting the silence percentage threshold to 20%, and the minimum distance to the last silence segment to approximately 14 frames (each of length 30ms) produced the best possible results. Interestingly, these are approximately almost the same values that [10] report. We therefore were using the same optimal values when comparing our algorithm with theirs, since both of our mechanisms share these optimal values for these constants.

Computing transcription accuracy is feasible, since most ASR APIs return information regarding the transcription accuracy confidence percentage. To compute timing accuracy, we calculated the percentage of transcriptions that had an absolute difference in timing, between showing up on the screen and the actual speech segment's occurrence, of less than 10ms. We show in Figure 5 a comparison of our algorithm's transcription accuracy results against [10]'s audio markup insertion algorithm, as well as against the original audio file's transcription accuracy results. In Figure 6, we show our comparison of both algorithms in terms of timing accuracy.

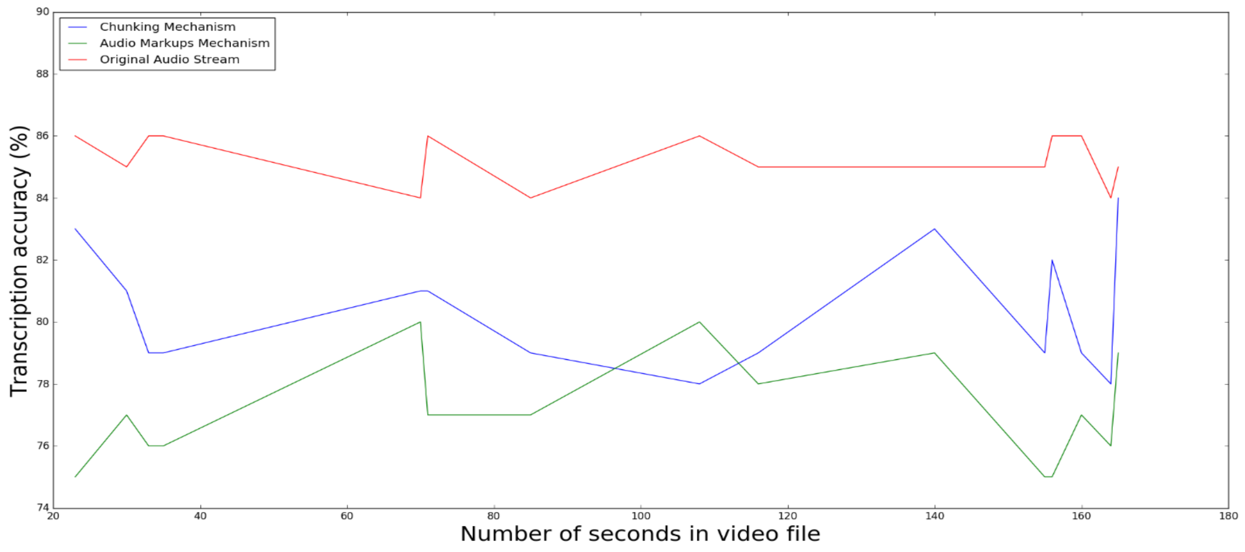


Figure 5: Transcription Accuracy (%) of our Speech Chunking algorithm, Markup Insertion algorithm, and Original File

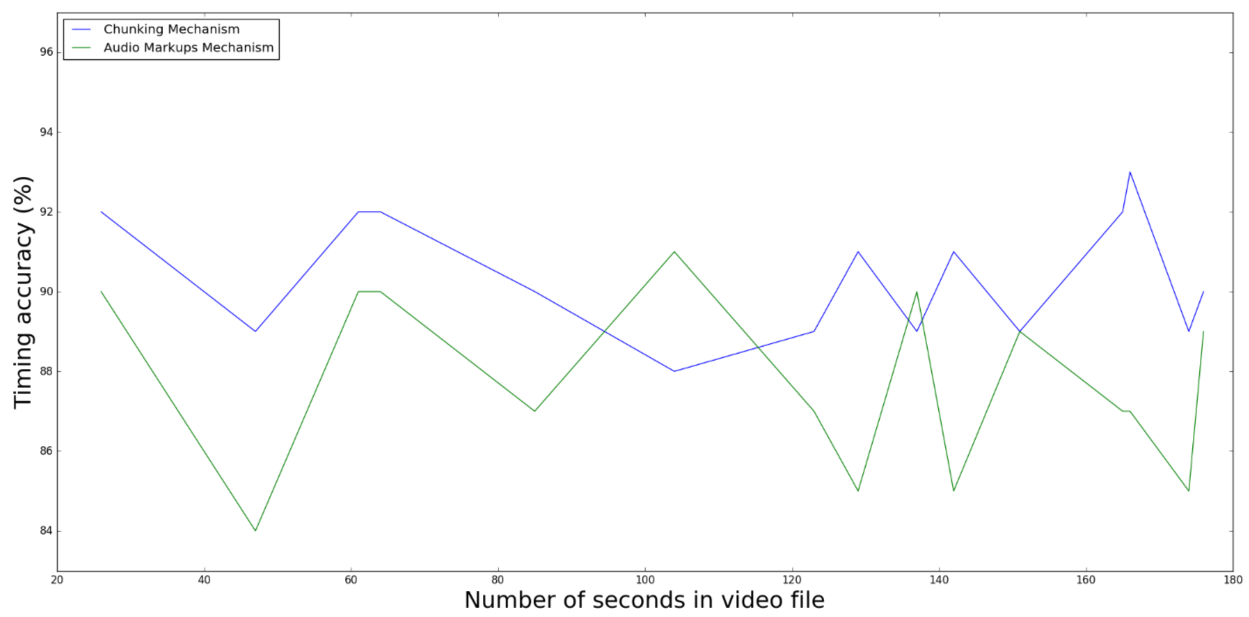


Figure 6: Timing Accuracy (%) of our Speech Chunking algorithm and Markup Insertion

From the transcription accuracy graph, we can see that our chunking mechanism performed slightly better than the markup insertion algorithm. Although both understandably did not achieve the transcription accuracy levels of simply transcribing the original audio file, we address the improvement of our chunking mechanism in Section 4.2. We can also notice that our mechanism performed slightly better than the markup insertion one in terms of timing accuracy as well.

5.2 Performance against YouTube's

Comparing the subtitles generated from our Android application and the ones generated from YouTube's close captioning engine, we were able to achieve better results both in terms of timing accuracy and transcription accuracy. However, because our mechanism takes a long amount of time to generate the subtitles (currently about one fourth of the total video file's running time), and because YouTube's infrastructure has to manage a copious amount of video data each second, our algorithm could not face the problem of scale faced by a video hosting site such as YouTube. Nonetheless, for the purposes of user generated videos on messaging platforms, and for short video files such as short news segments, our application does indeed do an apt job at automatically producing subtitles. Figure 8 shows in more detail our algorithm's performance against YouTube's in terms of timing accuracy, and Figure 9 shows a comparison of transcription accuracy. The same comparison techniques used in Section 5.1 are also used here. One main difference is that we had to calculate the transcription accuracy of YouTube's subtitles manually, since the output we obtain is merely the subtitle file, instead of accuracy information returned with the ASR APIs used in Section 5.1.

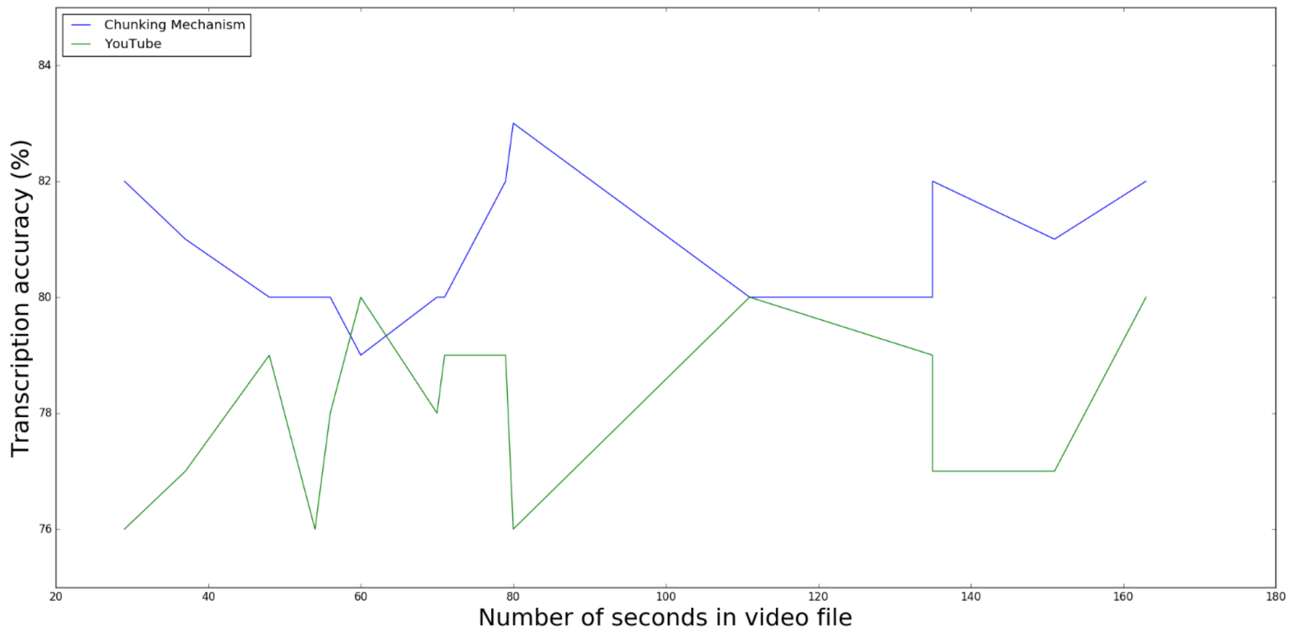


Figure 7: Transcription Accuracy (%) of our algorithm and YouTube'

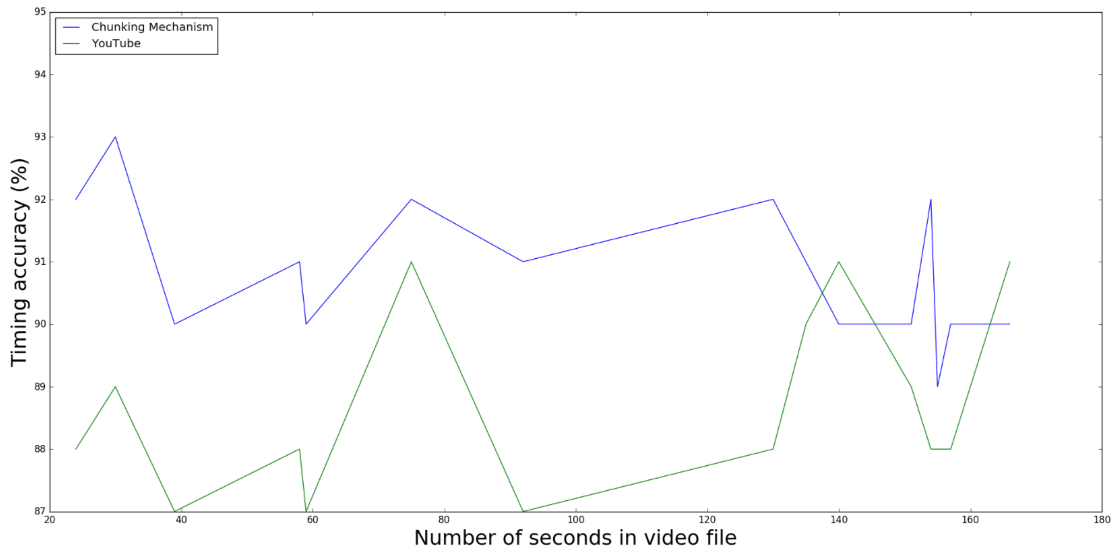


Figure 8: Timing Accuracy (%) of our algorithm and YouTube's

6. Future Work

The automation of subtitle generation and close captioning is still a growing field. Through our research in this paper, we have discovered that there exist potential work that could further improve subtitles and make them more widely attainable.

One of our current working topics is developing ways to make the algorithm faster. Currently, due to the chunking mechanism, the algorithm takes a fourth of the video's total running time in order to generate the subtitle file. While this is partly due to the chunking mechanism taking a significant amount of time, it is also due to the fact that we do not have the ASR API locally. In other words, each audio chunk has to be submitted through the internet, and its transcription is then sent back. For a lengthy video file, this could pose a significant amount of time. While our mechanism produces better results than YouTube's close captioning algorithm, it is definitely slower, something we plan to address in future work.

More importantly, our benchmarking framework is planned to be used in order to continuously hone the constants used in our algorithm. As [10] note, there could be a user interface where the user chooses the type of video they are passing into the application (i.e. colloquial video message, short news segment, etc.), and based on this information, the application can use the optimal constants. Indeed, after the obtainment of user feedback, statistical analysis on the dataset could be implemented in order to discover which ASR APIs work best with specific types of speech, the optimal constant values, etc. This data would, in turn, be used in the interface that [10] propose.

Another future upgrade we are considering is the possibility of producing a real-time transcription mechanism of all input audio streams on the phone. This basically means that the application, when turned on, would be continuously processing audio input to the phone, submitting it to an ASR API, and printing the transcription to the screen, mostly using an interface similar to Facebook chat heads.[11] This upgrade will enable the application's functionality to expand exponentially. In real time, as long as the application is active, any audio stream will be sent over to an ASR API for transcription. Therefore, the transcription would not be constricted to offline video files on a phone's memory card, and the user will not have to ask the application to generate the subtitles first. One application that uses real time transcription is RogerVoice, a VoIP calling application.[12] While the application is commercial and only transcribes phone call streams instead of any general audio stream on the phone, it is still a large step in the right direction in order to make close-captions and transcriptions as easily accessible as possible.

[11] https://www.facebook.com/help/messenger-app/1611232179138526
[12] https://rogervoice.com/

7. Conclusion

In this paper, we have proposed an Android application that provides automatically generated subtitles that can be of major help to those with hearing impairments and to language learners. Because the application was developed on Android, and because the growing trend of using smartphones and tablets for video consumption, this application has the potential to reach a wide array of individuals. We have proposed a speech chunking mechanism that has proved to provide better results than YouTube's native closed-captioning functionality. Moreover, our chunking mechanism has proven to maintain speech transcription accuracy, as opposed to the insertion of audio markups approach that was discussed in Section 2.3. We have also proposed a benchmarking framework for the constant improvement of the application's variables and for the study of which ASR services offer the best results with our model. Exploiting the ease of usage of smartphones and tablets is expected to make users much more prone to giving feedback. Finally, we have outlined future work needed in order to make subtitles and close captions even more accessible to users who need them.

8. Honor Code

This paper represents my own work and is in accordance with Princeton University's Honor Code.

Ragy Morkos

9. References

[1] Ronchetti, M. (2010). Using video lectures to make teaching more interactive. *iJET*, *5*(1), 45-48.
[2] Brecht, H. D. (2012). Learning from online video lectures. *Journal of Information Technology Education*, *11*, 227-250.
[3] O'Hara, K., Mitchell, A. S., & Vorbau, A. (2007, April). Consuming video on mobile devices. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 857-866). ACM.
[4] A. A. Ortiz, "Learning disabilities occurring concomitantly with linguistic differences," *Journal of Learning Disabilities*, vol. 30, no. 3, pp. 321–332, May/June 1997.
[5] Watkins, P. G., Razee, H., & Richters, J. (2012). 'I'm Telling You… The Language Barrier is the Most, the Biggest Challenge': Barriers to Education among Karen Refugee Women in Australia. Australian Journal of Education, 56(2), 126-141.
[6] Garza, T. J. (1991). Evaluating the use of captioned video materials in advanced foreign language learning. Foreign Language Annals, 24(3), 239-258.
[7] Wald, M. (2008). Learning through multimedia: Speech recognition enhancing accessibility and interaction. Journal of Educational Multimedia and Hypermedia, 17(2), 215.
[8] Zanón, N. T. (2006). Using subtitles to enhance foreign language learning. *Porta Linguarum: revista internacional de didáctica de las lenguas extranjeras*, (6), 4.
[9] Ivarsson, J. (2009). The History of Subtitles in Europe. *Dubbing and subtitling in a world context*, 3-12.
[10] Federico, M., & Furini, M. (2014). An automatic caption alignment mechanism for off-the-shelf speech recognition technologies. Multimedia tools and applications, 72(1), 21-40.
[11] Kim SK, Hwang DS, Kim JY, Seo YS (2005) An effective news anchorperson shot detection method based on adaptive audio/visual model generation. In: Proceedings of the international conference on image and video retrieval (CIVR), pp 276–285

10. Appendix

Appendix A

Pseudo code for Speech Chunking Algorithm

```
/*** Important constants used in our program ***/
static final int number_of_APIs = a; // number of speech-to-text APIs available in our system
static int audio_samples_millisecond; // usually found in an audio file's header information
static final double size_frame = 30.0; // usual size of an audio frame
static number_frames; // this is the number of virtual audio frames composing the audio stream
static int used_API = Random.nextInt(number); // random selection of an API
static double[] sound_energy; // array of doubles that will store the audio energy of each frame
static double silence_threshold_percentage = b; // used silence_threshold_percentage
static double silence_threshold; // silence threshold for this video
static final int min_speech_segment = c; // minimum number of frames needed for word utterance
static final int min_distanceTo_silence = d; // minimum number of frames needed between silent frames
static final int consecutive_silence_threshold = e; // minimum number of consecutive silence frames
static final int offset = (int)(consecutive_silence_threshold / 2);

/* Function Declarations. These are all the functions used in the main body of our program. */

// Function that initializes all uninitilaized variables above and
// returns audio file of the stream for use with the speech-to-text API
Audio Initializer(String VideoFileName)
{
   Audio file = // Extract audio from video

   int length = // obtain the duration, in milliseconds, of audio file from file's header info
   int last_frame = length % size_frame; // obtain the size of the last frame
   int audio_samples_millisecond = // obtain this number from audio file's header info
   int number_frames = Math.ceil(length / size_frame); // obtain number of audio frames

   // initialize sound energy array and calculate energy for each frame
   sound_energy = new double[number_frames];
   for(int i = 0; i < number_frames - 1; i += 1)
   {
      // sum the energy level of all the audio samples in this audio frame
      double sum = 0;
      int N = audio_samples_millisecond * 30;
      for(j = 0; j < N; j += 1)
      {
         sum += // add sound energy of this audio sample
      }

      // take the square root of this sum and divide it by N. This
      // will be the array entry in the sound_energy array
      sound_energy[i] = Math.sqrt(sum) / N;
   }
   // calculate energy level of last frame
   double sum = 0;
   int N = audio_samples_millisecond * last_frame;
   for (int i = 0; i < N; i += 1)
   {
      sum += // add sound energy of this audio sample
   }
   // take the square root of this sum and divide it by N. This
   // will be the array entry in the sound_energy array
   sound_energy[number_frames - 1] = Math.sqrt(sum) / N;

   // calculate silence_threshold
   double[] temp = double[number_frames];
   for (int i = 0; i < number_frames; i += 1)
   {
      temp[i] = sound_energy[i];
   }
   temp.sort();
   silence_threshold = temp[(int)silence_threshold_percentage * number_frames];

   // finally, return audio file of the video's audio stream
   return file;
}

// Functions for API's. Return transcribed text in form of String
String API0(Audio chunk)
{
   // String variable that will store transcription result
   String result;

   // Utilize the API to transcribe this chunk...

   return result;
}
String API1(Audio chunk)
{
   // String variable that will store transcription result
   String result;

   // Utilize the API to transcribe this chunk...

   return result;
}
String API2(Audio chunk)
{
   // String variable that will store transcription result
   String result;

   // Utilize the API to transcribe this chunk...

   return result;
}

// Function that utilizes the randomly chosen API at the beginning to transcribe the audio chunk
String SpeechToText(Audio chunk)
{
   // switch statement for producing the text using whatever API
   // was randomly chosen at the beginning of the program
```

```
    switch(used_API)
    {
        case 0:
            return API0(chunk);
        case 1:
            return API1(chunk);
        case 2:
            return API2(chunk);
    }
}

// Function that provides the .srt file for given mp3 argument.
// Return true for success and false for failure
boolean SubtitleGenerator(String AudioFileName)
{
    // Counter for speech segment (used in .srt file)
    int id = 1;

    // Make new file and append to it ".srt" extension
    PrintWriter writer = new PrintWriter(AudioFileName + ".srt", "UTF-8");

    /* go through sound energy array looking for speech segments */
    int i = 0;
    int last_silence = 0;
    int num_silent_frames = 0;
    while (i < number_frames)
    {
        // mark the beginning and ending of speech
        int beginning = i;
        int end = i;

        // As long as there is possible speech...
        while (num_silent_frames <= consecutive_silence_threshold)
        {
            if (sound_energy[i] <= silence_threshold)
            {
                num_silent_frames += 1;
            }
            else
            {
                num_silent_frames = 0;
            }
            end += 1;
            i += 1;
        }

        // mark the end of this silence sequence
        int this_silence = i;

        // If there was possible speech at least length of min_speech_segment...
        if ((end - index) > min_speech_segment && (last_silence - this_silence) > min_distanceTo_silence)
        {
            last_silence = this_silence; // reset last_silence

            Audio chunk = // obtain chunk of audio file correspoding to interval

            String result = SpeechToText(chunk); // obtain transcription of chunk

            /*** write that segment of speech to .srt file ***/

            // first write id of the speech segment
            writer.println(id);

            // convert from frames to milliseconds
            beginning *= size_frame;
            end *= size_frame;

            // calculate start time of this speech segment
            int hours1 = beginning / (60 * 60 * 1000);
            int minutes1 = (beginning / (60 * 1000)) % 60;
            int seconds1 = (beginning / 1000) % 60;
            int millseconds1 = beginning % 1000;

            // calculate end time of this speech segement
            end -= offset;
            int hours2 = end / (60 * 60 * 1000);
            int minutes2 = (end / (60 * 1000)) % 60;
            int seconds2 = (end / 1000) % 60;
            int millseconds2 = end % 1000;

            // write start and end times to .srt file
        writer.println(String.format("%02d", hours1)
           + ":" + String.format("%02d", minutes1)
           + ":" + String.format("%02d", seconds1)
           + "," + String.format("%03d", milliseconds1)
           + " --> " + String.format("%02d", hours21)
           + ":" + String.format("%02d", minutes2)
           + ":" + String.format("%02d", seconds2)
           + "," + String.format("%03d", milliseconds2));

        // write the transcription
        writer.println(string);
        writer.println();

        id += 1; // increase id for next subtitle

        del(chunk); // delete the temporary chunk Audio file
     }
  }

  // close writer (SRT) file
  writer.close();

  del(AudioFileName); // delete temporary mp3 file of audio stream

  return true;
}
```