RINGR Podcast Analyzer Tool

By

Bhuvaneshkumar Radjendirane
Elliot Couvignou
Sai Rajesh

Final Report for ECE 445, Senior Design, Spring 2020

TA: Shuai Tang

6 May, 2020

Project No. 70

Abstract

For this project we were sponsored by a podcast recording company, RINGR, to help create an audio editing tool specifically for the podcasting community. The main goal of this project was to automate a handful of audio editing processes based on the audio's transcription from speech to text. Transcription allows us to record all spoken words and the time they were spoken which allows us to give users power to edit words based on these timestamps. On top of this we can automate many tedious podcasting editing processes like censoring to further help the user save time. This form of audio editing is much more visually intuitive using words and word timestamps over the traditional method of visualization through waveforms and spectrograms. At the very least this project can be a supplementary editing software to help with repetitive editing processes to which the user then adds their own personal level of polish with traditional software.

Contents

1. Introduction

Our sponsor, RINGR, focuses on high-quality multi-agent podcast recording regardless of each person's separate location and compiles it all into one audio file with every speaker correctly in sync with each other [1]. An issue about podcast recording is that the amount of time to edit recordings takes too long and requires extensive knowledge of some audio editing software. This makes people who want to get into podcast recording have to be aware of the technicalities of editing audio which drives away a big audience. There are also many advanced editing processes, such as noise filtering, to further touch up their audio and achieve a desired level of polish. Since RINGR as a service helps with the initial step of recording the audio on individual channels, they have the potential to use audio processing with the raw audio to help bring down these technical barriers as well as saving time. Since podcasts are mostly speech reliant, we can relate most editing processes to transcripts containing words and when they were spoken. Modern day audio processing and speech recognition related products such as Amazon Alexa and Google Assistant proves that speech to text recognition is effective enough for use for this problem.

We created a tool that transcribes the recordings into a transcript of words and the word start and end times. From this we allow the user to edit the word timestamps to effectively move any words to any position in time and reflect these edits in the rendered result. We also provide a handful of extra features to show how transcription based editing can automate certain podcast editing processes such as pause shortening. All of these features are composed into one user interface (UI) where the only inputs are the audio recordings of each speaker. We plan to work with RINGR to further integrate this project into their current cloud-based model to be used by their customers.

1.1 Objective

The biggest goal of the project was to prototype and polish the workflow of editing podcast recordings through transcripts. On top of this we want to bring more features that serve to save the user editing time by performing common editing processes that can be coded through transcript words and their timestamps. All of these user timestamp edits and further features must allow the user to more conveniently analyze and rearrange audio segments on top of performing useful tedious editing techniques. There should also be a desire to use our project for its specific editing purposes over manually reproducing the same edits in traditional waveform editing software. On top of this the rendered audio must also be at a quality that is still more useful for the user to further change than the inputted recording.

2. Design

The main components of this project are the transcription system and the transcription user interface. The transcription system is simply the “under the hood” representation of the user’s audio data through waveforms, transcriptions, and other data structures to further minimize runtimes. This system is responsible for speech to text transcription of audio with word timestamps and all other editing services except the transcript editor which has its own system. The transcript user interface involves all the UI elements, the functionality of the transcript editor tab, and the rendering function that takes the results of all enabled features. This component is responsible for making the editing process as intuitive as possible for both visualization and the process of changing multiple word timestamps.

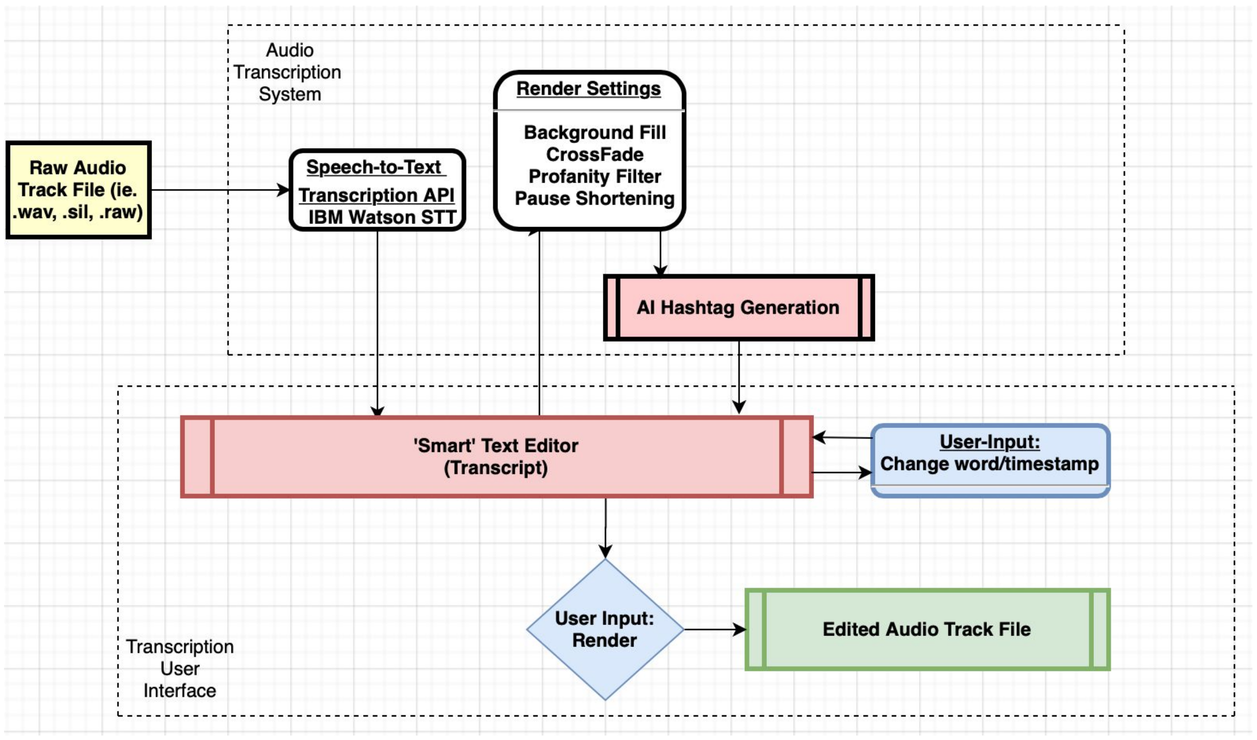
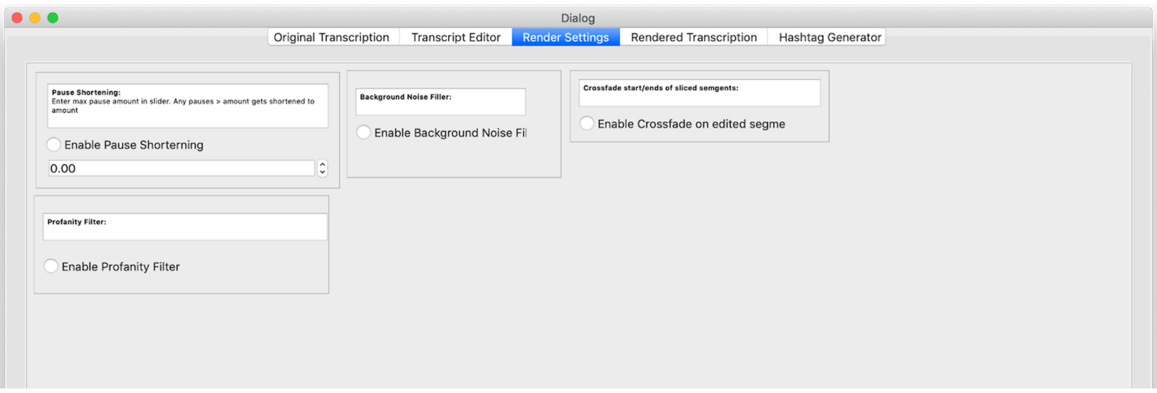

Audio Transcription System
Raw Audio Track File (ie. .wav, .sil, .raw)
Speech-to-Text Transcription API
IBM Watson STT
Render Settings
Background Fill
CrossFade
Profanity Filter
Pause Shortening
AI Hashtag Generation
'Smart' Text Editor (Transcript)
User-Input: Change word/timestamp
User Input: Render
Edited Audio Track File
Transcription User Interface

Figure 1. Diagram of our main components.

2.1 Transcription System

This component has a ton of smaller sub-components that each serve their own purpose in a significant way. The first use of this system is for transcribing the inputted audio into an array of words and an array for both the start and time timestamps for each word. These two data structures will be the data that later features will reference upon and edit as well. Other data structures are involved within the system for further optimizations but they will be addressed in their own sub-component section.



Pause Shortening:
Enter max pause amount in slider. Any pauses > amount gets shortened to amount

Enable Pause Shorterning

0.00

Background Noise Filler:

Enable Background Noise Fil

Crossfade start/ends of sliced semgents:

Enable Crossfade on edited segme

Profanity Filter:

Enable Profanity Filter

Figure 2. Render settings tab in UI

2.1.1 Transcription

For this component, we tested multiple different (STT) speech-to-text transcription APIs including Google's and CMU Sphinx because this component was central to the rest of our design. We settled on using IBM Watson STT because it offered multiple clear benefits over the possible candidate solutions. Most importantly, it offered the highest granularity of word timestamps (10 ms accuracy), which allowed us to have the highest accuracy possible in our transcription reliant features. It additionally offered hesitation word recognition, profanity recognition, and confidence levels for each word or phrase that was transcribed [2]. We used this model through the WebSocket interface allowing us to stream large audio files into the Watson STT interface, which is essential if podcasts are to be transcribed. The WebSocket interface allowed us to use a fully duplex TCP connection which was much more time efficient than using the regular Watson pipeline and allowed the transmission of large files to IBM [4].There was a high level of customizability offered in this IBM Watson STT model which made it the optimal choice for this project.

2.1.2 Crossfading

This feature, when enabled through the UI, performs a 75ms linear crossfade at the edges of audio segments that were edited. This is useful to reduce clicking and popping that results from poorly sliced audio segments that lack transients into their section. This feature tests if the words are connected immediately after each other so that only the edges of edited word segments are crossfaded instead of every word in the edited segments. Pseudocode for this function is presented below.

```
For each word in Transcript do
  if shifted[word] != 0
      if word.startTimestamp != previousWord.endTimestamp
         Crossfade left side of word
      if word.endTimestamp != nextWord.startTimestamp
         Crossfade right side of word
```

2.1.3 Background Noise Fill

This feature changes how rendering is done where any instances of absolute silence left from slicing out audio segments or extending the length of the audio length, is filled in with sampled background noise. We sample the background noise by looking at the inverse of the word transcript times. Any time range that doesn't overlap with times words were spoken are sampled as background noise. To fill in empty space with the background noise we short segments of the sample starting at random positions and smoothly crossfading between each short segment of noise.

2.1.4 Pause Shortening

Through this feature, the user is able to specify a maximum pause length they desire in their final output audio file. All pauses that are detected in the audio are cut down to this length if they are longer than the specified length and if they are shorter, they remain the same length. This is particularly useful for stitching together audio tracks that have disjoint sections or in other cases it can be used to trim the audio length to make it more compact. The pauses are detected by taking the inverse of the word timestamps, in other words, whatever audio is not transcribed is treated as a pause. These pauses are detected over the multiple different audio tracks/channels and the overlapping pause regions are used as the pause regions for the total audio file. Detecting the overlapping pause region is the first part of this feature's algorithm and it is also the most complex. As the number of channels/speakers increases, the runtime complexity of this part increases as well. This overlapping pause regions is what is cut down to the user-specified length. A visual depiction of how the overlapping pause region is calculated is shown below. The second part of the algorithm is where the list of overlapping pauses is used to cut the audio's pauses down to the specified length. This part is unaffected by the number of speakers and its runtime remains the same regardless of the number of speaker channels.

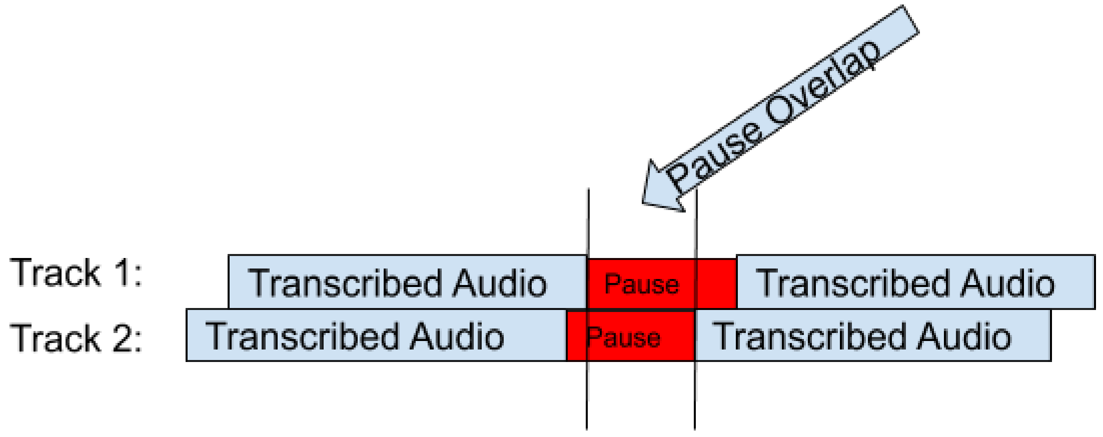


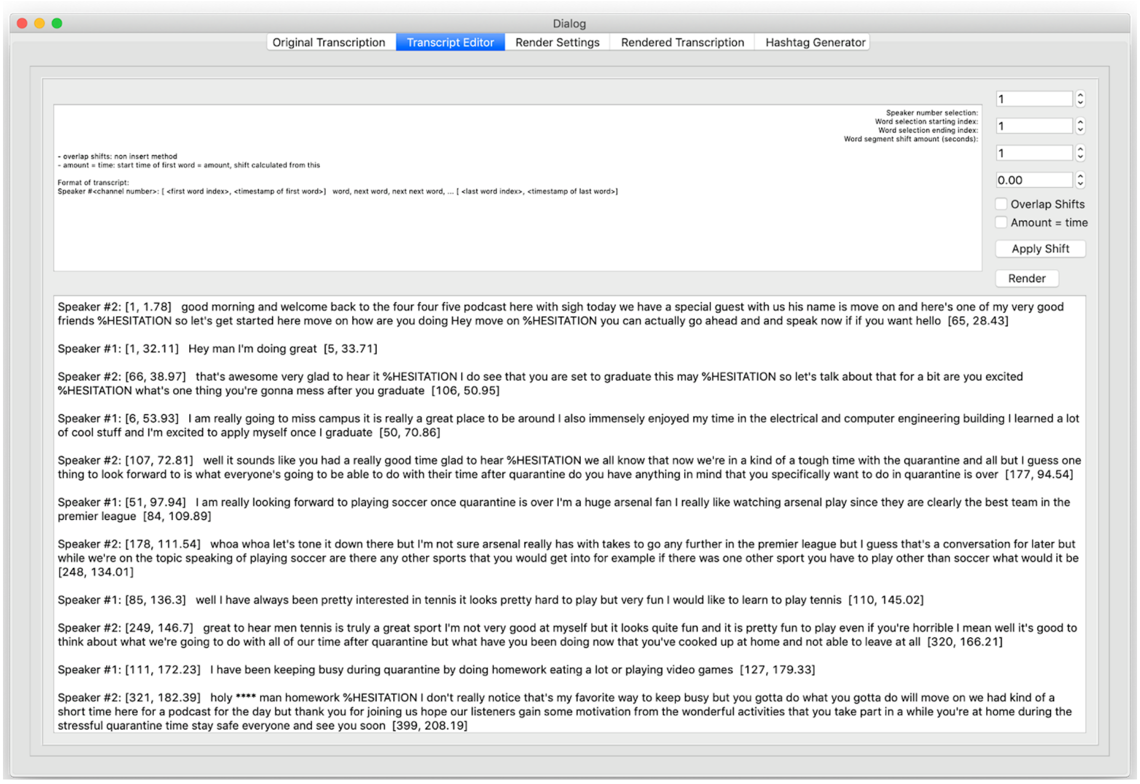
Figure 3. Pause Overlap Region Calculation Method

2.1.5 Profanity Filter

In addition to the features mentioned earlier, the user is also able to toggle a profanity filter on their input audio file(s). If switched on, the program will record the timestamps of the profane words detected by our IBM Watson STT model and apply a censor bleep equivalent to the length of the word in the appropriate areas. The program will traverse the audio file stitching together the non profane areas and when a profane word timestamp is reached, it will add a censor bleep of sufficient duration to the audio in place of the profane word.

2.2 Transcription Editor

This system combines all the other systems mentioned before into a UI system and includes the ability to manually edit word timestamps and render the overall result in an audio file. Our UI was built using pyQT5 which does lack a lot of visual appeal but allows us ease to prototype our features into one UI window. This user interface includes a couple tabs to help the user visualize their input, transcript and resulting outputs once rendering is done. One of the main tabs the user is going to spend a majority of their time on is the transcription editor tab. This tab allows the user to change timestamps of each word spoken to effectively rearrange every word spoken into any position the user wants. All transcription editing with the transcription editor and render features are taken into account and the resulting audio is created in the render sub-system.



Speaker number selection: 1
Word selection starting index: 1
Word selection ending index: 1
Word segment shift amount (seconds): 0.00

- overlap shifts: non insert method
- amount = time: start time of first word = amount, shift calculated from this

Format of transcript:
Speaker #<channel number>: [<first word index>, <timestamp of first word>]   word, next word, next next word, ... [<last word index>, <timestamp of last word>]

Speaker #2: [1, 1.78]   good morning and welcome back to the four four five podcast here with sigh today we have a special guest with us his name is move on and here's one of my very good friends %HESITATION so let's get started here move on how are you doing Hey move on %HESITATION you can actually go ahead and and speak now if if you want hello [65, 28.43]

Speaker #1: [1, 32.11]   Hey man I'm doing great [5, 33.71]

Speaker #2: [66, 38.97]   that's awesome very glad to hear it %HESITATION I do see that you are set to graduate this may %HESITATION so let's talk about that for a bit are you excited %HESITATION what's one thing you're gonna mess after you graduate [106, 50.95]

Speaker #1: [6, 53.93]   I am really going to miss campus it is really a great place to be around I also immensely enjoyed my time in the electrical and computer engineering building I learned a lot of cool stuff and I'm excited to apply myself once I graduate [50, 70.86]

Speaker #2: [107, 72.81]   well it sounds like you had a really good time glad to hear %HESITATION we all know that now we're in a kind of a tough time with the quarantine and all but I guess one thing to look forward to is what everyone's going to be able to do with their time after quarantine do you have anything in mind that you specifically want to do in quarantine is over [177, 94.54]

Speaker #1: [51, 97.94]   I am really looking forward to playing soccer once quarantine is over I'm a huge arsenal fan I really like watching arsenal play since they are clearly the best team in the premier league [84, 109.89]

Speaker #2: [178, 111.54]   whoa whoa let's tone it down there but I'm not sure arsenal really has with takes to go any further in the premier league but I guess that's a conversation for later but while we're on the topic speaking of playing soccer are there any other sports that you would get into for example if there was one other sport you have to play other than soccer what would it be [248, 134.01]

Speaker #1: [85, 136.3]   well I have always been pretty interested in tennis it looks pretty hard to play but very fun I would like to learn to play tennis [110, 145.02]

Speaker #2: [249, 146.7]   great to hear men tennis is truly a great sport I'm not very good at myself but it looks quite fun and it is pretty fun to play even if you're horrible I mean well it's good to think about what we're going to do with all of our time after quarantine but what have you been doing now that you've cooked up at home and not able to leave at all [320, 166.21]

Speaker #1: [111, 172.23]   I have been keeping busy during quarantine by doing homework eating a lot or playing video games [127, 179.33]

Speaker #2: [321, 182.39]   holy **** man homework %HESITATION I don't really notice that's my favorite way to keep busy but you gotta do what you gotta do will move on we had kind of a short time here for a podcast for the day but thank you for joining us hope our listeners gain some motivation from the wonderful activities that you take part in a while you're at home during the stressful quarantine time stay safe everyone and see you soon [399, 208.19]

Figure 4. Transcription editor in UI

2.2.1 Rendering

This sub-system is only activated through a user input by clicking on the "Render" button. Once activated, we begin rendering by reading all the changes done to the transcript and then feeding that information down to each render feature. Once the changes are done in a specific order we can begin piecing the resulting audio by looking at the changes in timestamps. The resulting audio is the same as the original but words with new timestamps should not be present in their old location and are instead present in their new, edited location. Pseudocode for this function is presented below.

```
//set output as input
render = audio
For each word in Transcript do
  if shifted[word] != 0 do
      word.newStart = word.start + shifted[word]
      word.newEnd = word.end + shifted[word]
      if word.newEnd*sampleRate > audio.length do
        // zero-pad extend and background noise fill on zero-pad if enabled

      // fill new timestamps with spoken word, remove from old spot
      render[word.newStart: word.newEnd] = audio[word.start: word.end]
      audio[word.start:word.end] -= audio[word.start:word.end]

      if sum(audio[word.start:word.end]) == 0 do
        // background noise fill if user enabled
return render
```

2.2.2 Hashtag Generation

This sub-system is only activated through a user input by clicking on the "Generate Hashtags" button located under the Hashtag Generator tab in the UI. Once activated, an information retrieval technique known as term frequency-inverse document frequency (TFIDF) begins on the latest version of the transcript, taking into account any edits already made by the user. The sub-system extracts the most high weighted terms, hereby referred to as keywords, in the transcript using TFIDF along with Python NLP library, nltk. . It then uses the spaCy NLP Python library in order to extract all keywords that are nouns. These nouns are used as suggested hashtags. In addition, the keywords are then used as search queries to find live hashtags used on social media platform, Twitter. The sub-system is then able to also provide to the user a list of Twitter-based hashtags related to the podcast topic. Therefore, when the "Generate Hashtags" button is pressed, the user is provided with 2 vast lists of suggested hashtags relating to the podcast topic.

3. Verification

Verification of these features and systems is mostly going to be focused on runtime as we tried our best to test our features to make sure they always work correctly, but this is hard to guarantee with our limited time. We always tested our features the best we could but at some point we found ourselves wasting time by trying to deliberately break our code to make sure every bug has been noticed. Overall all our features do perform in most cases although there might be some small edge cases that will result with minor artifacts in rendering.

3.1 Transcription System

The runtime for this system is too general to analyze and it is instead preferred to examine the runtime of the sub-systems features individually since the user chooses which features they want to include. It is important to note that the performance of all these features, except the transcription sub-system itself, rely on the word labeling accuracy and timestamp precision of the transcription service we use. All of these functions have been thoroughly tested although we haven't gone too far to really make sure every small bug is ironed out. Therefore the functions perform reliably although there might be some significant outliers that can be fixed once noticed.

3.1.1 Transcription

Our transcription model is the IBM Watson STT model, so the performance of this component is mostly out of our hands. Thankfully, we can measure the accuracy/confidence of this model directly as IBM provides this metric to us, ranging from the entire transcription level confidence all the way to a word level confidence. However, since we are guaranteed studio level audio quality from RINGR, our transcription confidence is assured to be very high. In terms of runtime, after much experimentation we found that IBM's model running through the WebSocket interface we set up takes approximately 0.5-0.75x the length of the input audio. For example, this translates to a runtime of about 2 minutes for an input audio file of 4 minute length just for the transcription alone.

Our transcription interface creates a data structure for each input audio file and stores relevant information that is used in the features of our interface. In terms of space complexity, our interface stores each word and its according timestamps (start and end timestamps). It also stores a copy of the audio file which is used to apply the edits the user specifies in the interface. This transcription data structure is the main storage unit for the information that is used by all of the transcription reliant features. This data structure is of linear space complexity as it only depends on the size of the audio file/number of words in the audio file.

3.1.2 Crossfading

The runtime of this feature adds a very minimal runtime of O(N), N = number of words, at the worst case and O(1) at the best case. The worst case scenario is extremely improbable and occurs when every word is changed and manually isolated apart from the others. This function required no new data structures to be added to our transcript class.

3.1.3 Background Noise Fill

The runtime for this feature is linear based on the length of silence left behind during rendering, O(S), S = length of silence. This length of silence left behind during each render depends on the type and amount of edits being done that allows these silences to occur. For example, if the user inputs a short sound file and edits one of their timestamps to be much longer than the input audio itself, then this function will run much longer than if the user outputs their audio length the same as the input. Performance and reliability of this function is not very good as there are multiple issues in the current implementation

With background noises being the inverse of word timestamps, we came into an issue where laughter and sighs were registered as background noise. This doesn't fit well with how the function is supposed to work and a solution is to instead lowpass filter the audio and spaces with relatively low energy can be categorized as background noise. This creates another problem of different users having varying levels of noise volume which is why that implementation doesn't exist in the final version.

3.1.4 Pause Shortening

Runtime for the pause shortening feature is dependent on the number of audio channels (speakers) and the length of the audio file. If we let 'M' denote the number of audio speaker channels and let 'N' denote the average number of words/length of audio per channel, then we can classify the runtime of this feature to be O(MN). In most cases, there are only two speakers in a given audio file, an interviewer and interviewee, so the runtime of this feature is in most cases approximately O(2N) ≅ O(N). If the input audio file is a group conversation between multiple speakers, then the pause overlap detection algorithm becomes much more complex and will take a much longer time to run, therefore increasing the total runtime of this feature. Due to this feature only relying on pre-existing data structures (the transcription data structure), it adds no space complexity to the overall interface.

3.1.5 Profanity Filter

The overall runtime for the profanity filter toggle is O(N), N being the length of the audio file, as the algorithm traverses the audio file once applying a censor bleep at the timestamp ranges of the profane words detected by the IBM Watson STT model. It also requires no new data structures, as it retrieves the program-relevant data from the transcription data structure which is created when the interface is started up, therefore adding no space complexity.

3.2 Transcription Editor

Verification for this system is going to be focused on runtimes for the transcript editor and the rendering function. Since our UI was built using pyQT5, we can be confident that code related to this library is sturdy and quick as this is a widely used library. The transcription editor handles inserts very quickly and the runtime scales linearly with the amount of words that the user wants to shift. The runtime is so quick that there is barely any noticeable delay for applying shifts until the user wants to shift over 50 words. The amount that the words are shifted have no effect on the runtime of editing timestamps, only the amount of words that are edited.

3.2.1 Rendering

Rendering is the longest running function in the entire project and it's for good reason. This simply loops through every word and checks whether the timestamps changed from its original values. From this the runtime of rendering scales linearly with the amount of words with timestamps that differ from their original values, O(W), W = number of words edited. Other rendering features do affect this runtime as those features change word timestamps which are then handed down to the render function once they are finished. These other functions only change timestamp values so the behavior of the runtime is still linear, the difference is that more edited words need to be worked through. This function is very stable since it was the first function to be added which gave it a lot of time to be figured out completely. The only unseen bugs that could come up are more likely to be bugs related to other features that have been carried over.

3.2.2 Hashtag Generation

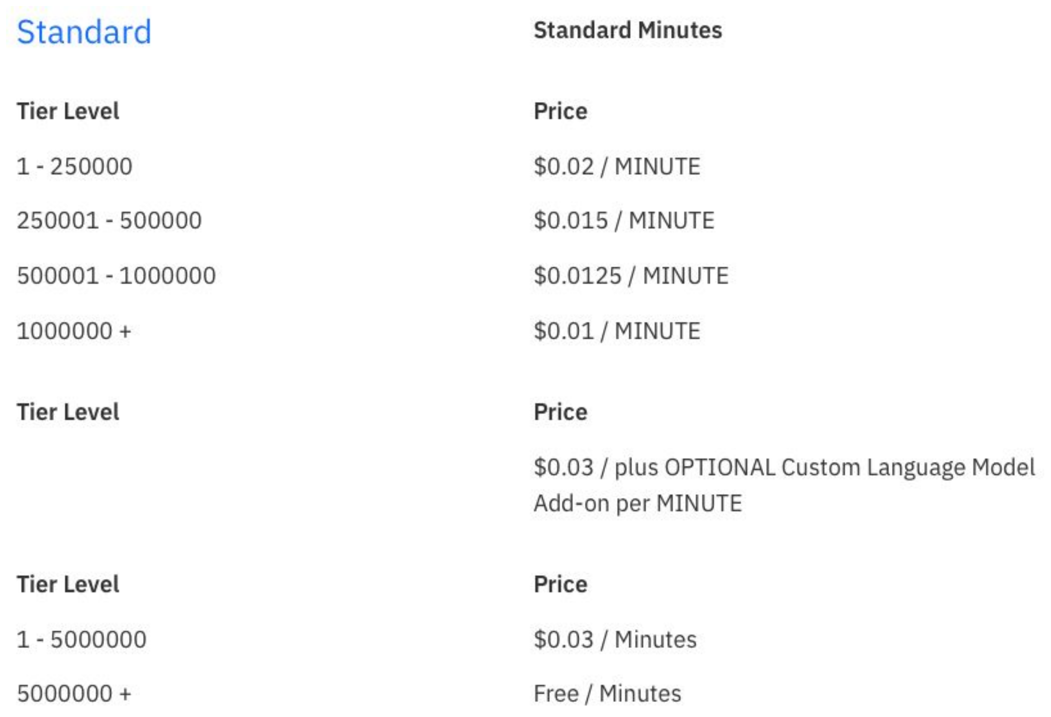
Unlike most other subsystems involved in the project, the hashtag generation feature is based mostly on NLP techniques and TFIDF. In order to generate hashtags based on transcription text, keywords were first extracted from the transcript using TFIDF. Prior to doing that however, the text must be cleaned and vetted properly [5]. All words are made lowercase, all numbers are removed from the text, stop words are removed and all words are lemmatized. Then TFIDF is run on the cleaned text, providing keywords of the text. However when testing, it was concluded that many of these words were not hashtag-worthy. As a result it was decided that only nouns would be extracted for the purposes of creating hashtags. At first as a solution, nltk's part-of-speech (POS) tagging was used in order to extract the nouns. However, this library was not properly detecting the POS of words in the transcript since its training data was not relative to our podcast use case. As a result, Python's spaCy NLP Library was used instead of nltk for POS tagging. More specifically, the 789 MB "en_core_web_lg" multi-task CNN model by spaCy was used and found to be the most effective POS tagging technique, providing hashtags with noun POS 100% of the time.

4. Costs

As this project was completely software-based, with no need for hardware components of any sort, our cumulative costs were kept quite low. The final cost of the project consists of only a service cost and a labor cost as detailed in the following sections, 4.1 and 4.2.

4.1 Services

In order to successfully retrieve transcriptions of inputted audio podcasts, IBM Watson’s Speech-To-Text API was used. We initially used the free trial service referred to as the IBM Lite model, but this only allows for 500 minutes of use. After exceeding this limit, we switch to the IBM Standard model which is a paid SaaS. The full pricing metrics for this service are listed below.



Standard	Standard Minutes
Tier Level	**Price**
1 - 250000	$0.02 / MINUTE
250001 - 500000	$0.015 / MINUTE
500001 - 1000000	$0.0125 / MINUTE
1000000 +	$0.01 / MINUTE
Tier Level	**Price**
	$0.03 / plus OPTIONAL Custom Language Model Add-on per MINUTE
Tier Level	**Price**
1 - 5000000	$0.03 / Minutes
5000000 +	Free / Minutes

Table 1. IBM Watson STT Standard model pricing [3]

After using all 500 initial minutes provided in the Lite Version, we upgraded to the Standard version and spent a total of $27.30 during the entire course of developing and training our software. In the possibility this project is to be made commercially viable, service costs would increase as the software would need to run on more reliable and large servers in order to handle the capacity of users. This would effectively increase service cost, in turn increasing overall cost.

4.2 Labor

The project was completed by three individuals, all members of Team 70. Complete labor costs, assuming every member of the group is given an equal amount, is given by:

$$LABOR = (\$50 / hour) * 2.5 * (160 \text{ hours total}) = \$20{,}000$$

Therefore, with 3 members the total Labor Cost would be given by:

$$TOTAL\ LABOR = \$20{,}000 * 3 = \$60{,}000$$

The total Labor Cost estimate of the project is $60,000. Therefore, the total cost of this project over the course of the semester can be summed up as follows:

$$TOTAL\ COST = \$60{,}000 + \$27.30 = \$60{,}027.30$$

5. Conclusion

5.1 Executive Summary

Over the course of this project a simple and intuitive alternative to waveform-based podcast editing was developed. Using our podcast editing tool available with essential features such as Pause Shortening, Crossfading, Profanity Filtering and more, RINGR users will be able to efficiently and easily edit their own podcast content in a matter of minutes. Users will also be able to speed up the process of posting their produced final podcasts to various social media platforms with the help of additional features such as Hashtag Generation.

5.2 Going Forward

With the preliminary features created, we will look to merge them completely with RINGR's application in order to allow RINGR users to gain access to them. This project has also opened multiple new avenues for feature possibilities. In the near future we will be working with RINGR in order to finetune established features mentioned earlier in this paper, as well as create new ones for RIGNR users. Some of the suggested features we will be continuing with include:

- Synopsis Generation to automatically create summaries for edited podcasts.
- Maturity Ratings for podcasts depending on degree of profanity.
- Options to insert sound effects into podcasts.
- Filters for hesitation fill words
    - i.e. 'Um' and 'Ah'
- Increase capability to render multiple features simultaneously

6. References

[1] ringr.com, “How It Works”2020. [Online]. Available:
https://www.ringr.com/. [Accessed: 2-Feb-2020]

[2] “Speech-To-Text IBM Cloud API,” IBM Cloud. [Online]. Available:
https://cloud.ibm.com/apidocs/speech-to-text. [Accessed: 06-May-2020].

[3] “Watson Speech to Text - Pricing,” IBM. [Online]. Available:
https://www.ibm.com/cloud/watson-speech-to-text/pricing. [Accessed: 07-May-2020].

[4] “The WebSocket API (WebSockets),” MDN Web Docs. [Online]. Available:
https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. [Accessed: 07-May-2020]

[5] K. Y. Abeywardana, G. A. R., H. N., S. H. P., and T. T. M. N. S., “Hashtag Generator and Content Authenticator.” Available:
“https://pdfs.semanticscholar.org/cd77/0aa69001d53693fc247fac2a75a6b8ebea8c.pdf”.

7. Appendix A [Requirements and Verification Table]

Requirement	Verification	Verification Status
Must be able to return word timestamps within 100ms of accuracy	A. Transcribe a portion of audio with known correct transcription. B. Select a few words at random from the transcription and obtain timestamps for the word from the API. C. Check a spectrogram of the audio at the obtained timestamps and check accuracy of when the final sounds of the words chosen end.	Yes
Must be accurate in recognizing podcast subjects at least 90% of the time.	A. Compile podcast samples on known subjects. B. Run Algorithm on the audio files compiled and ensure at least 90% of returned podcast subjects match with initial list.	Yes
Must accurately detect 85% of instances of 'word' [ex: 'um' or 'uh'] that the user wants removed from the transcript.	A. Transcribe at least 30 minutes of RINGR audio. B. Save initial transcription [pre-edited] to a text file. C. Apply filter and specify sound to be removed. D. Check final [post-editing] transcript and compare the number of instances removed to number of instances initially in saved transcript. Accuracy should be above 80%.	Yes
Post-edit audio rendering should not sound fragmented or have words cut off.	A. Transcribe 10 minutes of audio and create spectrogram of original audio file for later comparison. B. Move around sentences and words in the smart text editor and obtain new spectrograms. C. Compare moved portions in new spectrogram to original spectrogram and ensure no words were fragmented in the process.	Yes