NLP Review 3

Shyam Nair - 19BCE1017, Akash A - 19BCE1536

Abstract

Due to the ongoing pandemic, most of the Institutions are conducting their classes in online mode. Because of this students are having issues like power cuts, network issues, illness due to covid, etc. are unable to attend classes and thereby they are mostly dependent on the lecture recordings. Due to lack of time they are unable to go through the complete recordings and revise.

Our tool aims to solve this problem but summarizing the lectures into a text summary and compressed video format. This summarized format will help the students save time as the time required to go through them would be a lot shorter than before. This will especially help during the exams when the students are expected to go through a lot of content in a short span of time. We'll also try to provide some analysis of the lecture along with the summary.

1 Title

Lecture Summarizer and analysis

2 Modules

- Audio Extraction.
- Transcription of audio.
- Text Summarization.
- Mapping Summarized text to timestamps.
- Video Compression
- Cloud.
- UI for the tool.

3 Motivation

Most students perpetually dace a lack of time in their academic carrier. In such a scenario, a lecture summarizer can be of great use as it will help reduce the time required by a student to revise or study from a recorded lecture. Also, according to a study, most adults have a attention span of 25-30 minutes. So, our lecture summarizer will help those who suffer from short attention spans.

4 Objectives

- To create a tool that will take a video lecture as input and provide a summarized version of the video along with the summarized transcript of the video.
- Make use of NLP techniques to extract the top n sentences to be considered for the summarized version.
- The system should be able to generate a summarized video corresponding to the summarized transcript

5 Methodlogy

- The user will be required to upload a video/Lecture of whose the summary is required using our interactive UI
- In order to transcribe the video, we will first extract the audio file from the uploaded video file.

- Using Google speech-to-text API we would be transcribing the audio file.
- Then we would be mapping each word to the corresponding timestamp it was spoken in the video.
- Text summarization
  - We will first perform the required pre-processing
  - Generation of cosine-similarity matrix across sentences.
  - We would be ranking the sentences in the matrix, then sort and pick the top sentences.
- Video Compression
  - Based on the timestamps of the summarized sentences we would be trimming the original video using moviepy library in python.
- Then we would be connecting our colab notebook to the front-end using xyz cloud. The front-end will help the user to upload the video and access the text and video summary along with the analysis.

6 Literature Survey

1. Juan Manuel Torres Moreno has come up ARTEX - Another Text Summariser. In order to rank sentences, a simple inner product is calculated between each sentence, a document vector (text topic) and a lexical vector (vocabulary used by a sentence). Summaries are then generated by assembling the highest ranked sentences. No ruled-based linguistic post-processing is necessary in order to obtain summaries.

2. Nikhil Shirwandkar, Samidha Kulkarni designed an approach for extractive text summarisation using deep learning. In this work, an approach for Extractive text summarization is designed and implemented for single document summarization. It uses a combination of Restricted Boltzmann Machine and Fuzzy Logic to select important sentences from the text still keeping the summary meaningful and lossless.

3. Surabhi Adhikari, Rahul and Monika presented a NLP based approach for text summzrization using ML algorithms. The paper mostly discusses about the structured based and semantic based approaches for summarization of the text documents

4. Adhika Widyassari, Supriadi Rustad, Guruh Fajar Shidik have done a review of automatic text summarization techniques and methods. This paper provides a broad and systematic review of research in the field of text summarization published from 2008 to 2019.

5. Veronica Morfi, Dan Stowell a system for audio transcription on low-resource datasets using deep learning. In this paper, they propose factorising the final task of audio transcription into multiple intermediate tasks in order to improve the training performance when dealing with this kind of low-resource datasets. The authors have evaluated three data-efficient approaches of training a stacked convolutional and recurrent neural network for the intermediate tasks.

6. Miguel Negrao, Patricio Dominigues designed SpeechToText: An open-source software for automatic detection and transcription of voice recordings in digital fornsics. SpeechToText achieves 100% accuracy for detecting voice in unencrypted audio/video files, a word error rate (WER) of 27.2% when transcribing English voice messages by non-native speakers and a WER of 7.80% for the test-clean set of LibriSpeech.

7. Mrigank Rochan, Linwei Ye and Yang Wang have proposed video summarization using fully convolutional sequence networks. Unlike existing approaches that use recurrent models, we propose fully convolutional sequence models to

solve video summarization. We firstly establish a novel connection between semantic segmentation and video summarization, and then adapt popular semantic segmentation networks for video summarization.

7 Speech-to-text

It all starts with human sound in a normal environment. Technically, this environment is referred to as an analog environment. A computer can't work with analog data; it needs digital data. This is why the first piece of equipment needed is an analog to digital converter.

7.1 Analog to Digital converter

A microphone usually serves as an analog to digital converter. The conversion can be visualized in a graph known as a spectrogram. To create a spectrogram, three main steps are followed:

The sound wave is captured and placed in a graph showing its amplitude over time. Amplitude units are always expressed in decibels (dB). The wave is then chopped into blocks of approximately one second, where the height of a block determines its state. Each state is then allocated a number hence successfully converting the sound from analog to digital. Even when the data is digitized, something is still missing. In the speech recognition process, we need three elements of sound. Its frequency, intensity, and time it took to make it. Therefore, a complex speech recognition algorithm known as the Fast Fourier Transform is used to convert the graph into a spectrogram.

7.2 Linguistics and Phonemes

A phoneme is a distinct unit of sound that distinguishes one word from another in a particular language. It is the smallest part of a word that can be changed – and, when changed, the meaning of the word is also changed. For instance, the word "thumb" and the word "dumb" are two different words that are distinguishable by the substitution of the phoneme "th" with the phoneme "d."

Phonemes can be spoken differently by different people. Such variations are known as allophones, and they occur due to accents, age, gender, the position of the phoneme within the word, or even the speaker's emotional state.

Phonemes are important because they are the basic building blocks used by a speech recognition algorithm to place them in the right order to form words and sentences. Speech recognition does this using two techniques – the Hidden Markov Model and Neural Networks.

7.3 Hidden Markov Model in Speech Recognition

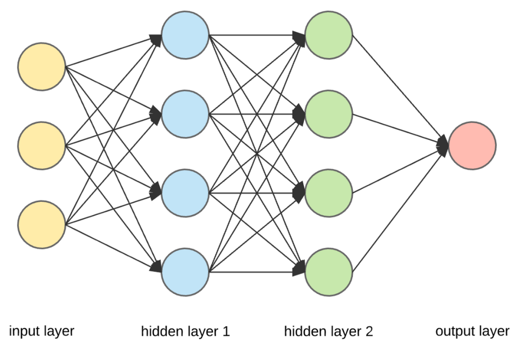
The Hidden Markov model in speech recognition, arranges phonemes in the right order by using statistical probabilities. To do this, it uses three different layers.

In the first layer, the model has to check the acoustic level and the probability that the phoneme it has detected is the correct one. As stated before, the variation of phonemes depends on several different factors, such as accents, cadence, emotions, gender, etc.

In the second layer, the model checks phonemes that are next to each other and the probability that they should be next to each other. For example, if you have the sound "st," then most likely a vowel such as "a" will follow. It's less likely or even impossible for an "n" phoneme to follow an "st" phoneme – at least in the English language.

Finally, in the third layer, the model checks the word level. That is, whether words next to each other make sense. It does this by checking the probability that they should be next to each other. For example, it will check if there are too many or too few verbs in the phrase. It also checks adverbs, subjects, and several other components of a sentence.

7.4 Artificial Neural Networks



A neural network is a network of nodes that are built using an input layer, a hidden layer composed of many different layers, and an output layer.

The connections all have different weights, and only the information that has reached a certain threshold is sent through to the next node. If a node has to choose between two inputs, it chooses the node's input with which it has the strongest connection. In some systems, it can also take both inputs and come up with a ratio.

The advantage of neural networks is that they are flexible and can, therefore, change over time. This means that the neural network has to be trained as all the different connections initially have the same weight. Input is given to the neural network, and the desired output specified. The neural network then does its thing and comes up with a certain output that is not the same as the desired output because more training is needed. This difference is the error. The neural network understands that there is an error and therefore starts adapting itself to reduce the error. For the neural network to keep improving and eliminate the error, it needs a lot of input.

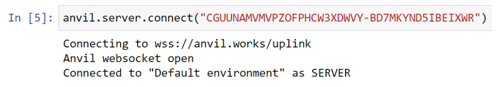
The weaknesses of Neural Networks are mitigated by the strengths of the Hidden Markov Model and vice versa. This is why the Hidden Markov Model and Neural Networks are used together in speech recognition applications.

8 Extractive Text Summarization

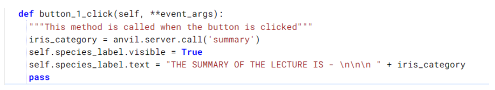
The extractive text summarization approach involves picking up the most important phrases and lines from the documents. It then combines all the important lines to create the summary. So, in this case, every line and word of the summary actually belongs to the original document which is summarized. Extractive methods attempt to summarize articles by selecting a subset of words that retain the most important points.This approach weights the important part of sentences and uses the same to form the summary. Different algorithm and techniques are used to define weights for the sentences and further rank them based on importance and similarity among each other. Input document $\rightarrow$ sentences similarity $\rightarrow$ weight sentences $\rightarrow$ select sentences with higher rank. We are going to use cosine similarity as a metric of similarity between sentences. Cosine similarity is defined as a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them. Since we will be representing our sentences as the bunch of vectors, we can use it to find the similarity among sentences. Its measures cosine of the angle between vectors. Angle will be 0 if sentences are similar.

9 PROPOSED WORK

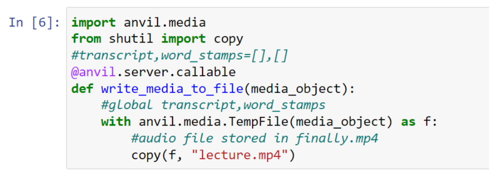
1. The first part of the system is to fetch the video uploaded by the user to the backend. This is video that will be processed later on to generate summary. We will be making this happen using Anvil.



```
In [5]: anvil.server.connect("CGUUNAMVMVPZOFPHCW3XDWVY-BD7MKYND5IBEIXWR")

        Connecting to wss://anvil.works/uplink
        Anvil websocket open
        Connected to "Default environment" as SERVER
```

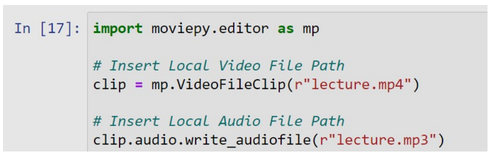


```
def button_1_click(self, **event_args):
  """This method is called when the button is clicked"""
  iris_category = anvil.server.call('summary')
  self.species_label.visible = True
  self.species_label.text = "THE SUMMARY OF THE LECTURE IS - \n\n\n " + iris_category
  pass
```



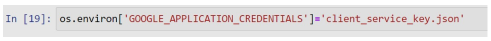
```
In [6]: import anvil.media
        from shutil import copy
        #transcript,word_stamps=[],[]
        @anvil.server.callable
        def write_media_to_file(media_object):
            #global transcript,word_stamps
            with anvil.media.TempFile(media_object) as f:
                #audio file stored in finally.mp4
                copy(f, "lecture.mp4")
```

2. The next part of processing the video involves extracting the audio from the video file. This is done using the moviepy library containing the clip.audio.write_audiofile() function.



```
In [17]: import moviepy.editor as mp

         # Insert Local Video File Path
         clip = mp.VideoFileClip(r"lecture.mp4")

         # Insert Local Audio File Path
         clip.audio.write_audiofile(r"lecture.mp3")
```

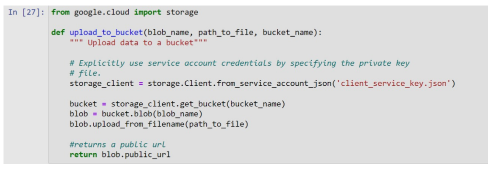
3. Then we are going to set up the environment for using google speech-to-text api. This involves passing client_service_key.json file to the environment variable 'GOOGLE_APPLICATION_CREDENTIALS'.



```
In [19]: os.environ['GOOGLE_APPLICATION_CREDENTIALS']='client_service_key.json'
```

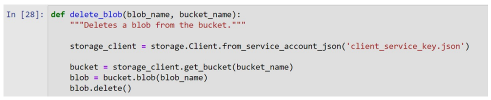
4. Transcribing the audio file:

   - The first part is to create cloud storage bucket to upload the video into. This is done using blob name, path to file, bucket name



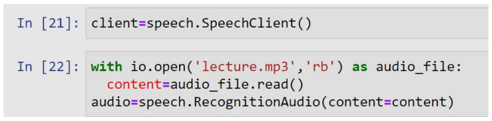
```
In [27]: from google.cloud import storage

         def upload_to_bucket(blob_name, path_to_file, bucket_name):
             """ Upload data to a bucket"""

             # Explicitly use service account credentials by specifying the private key
             # file.
             storage_client = storage.Client.from_service_account_json('client_service_key.json')

             bucket = storage_client.get_bucket(bucket_name)
             blob = bucket.blob(blob_name)
             blob.upload_from_filename(path_to_file)

             #returns a public url
             return blob.public_url
```

   - Once the video is uploaded and transcribed, we don't the the video to be stored on cloud anymore, so we can delete the blob. This is done using blob.delete() function.

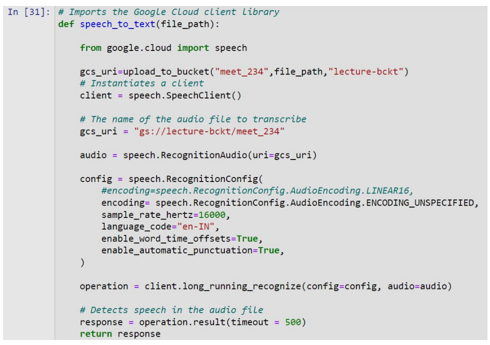


```
In [28]: def delete_blob(blob_name, bucket_name):
             """Deletes a blob from the bucket."""

             storage_client = storage.Client.from_service_account_json('client_service_key.json')

             bucket = storage_client.get_bucket(bucket_name)
             blob = bucket.blob(blob_name)
             blob.delete()
```

   - Now we can transcribe the audio file. Before transcribing the audio, we need to provide the configuration for the transcription. This involves initializing the encoding type, sample rate in hertz, language code and also set the flags enable_word_time_offsets, enable_automatic_punctuation to true.

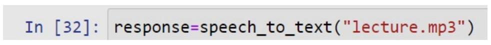


```
In [21]: client=speech.SpeechClient()

In [22]: with io.open('lecture.mp3','rb') as audio_file:
           content=audio_file.read()
         audio=speech.RecognitionAudio(content=content)
```



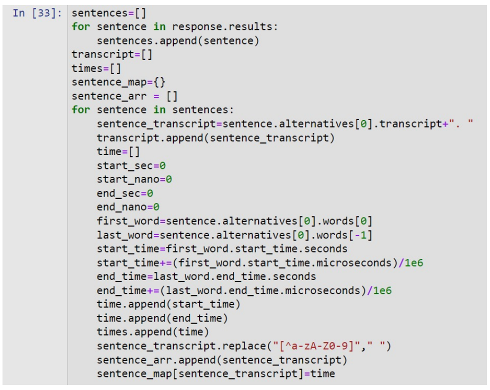
```
In [31]: # Imports the Google Cloud client library
         def speech_to_text(file_path):

             from google.cloud import speech

             gcs_uri=upload_to_bucket("meet_234",file_path,"lecture-bckt")
             # Instantiates a client
             client = speech.SpeechClient()

             # The name of the audio file to transcribe
             gcs_uri = "gs://lecture-bckt/meet_234"

             audio = speech.RecognitionAudio(uri=gcs_uri)

             config = speech.RecognitionConfig(
                 #encoding=speech.RecognitionConfig.AudioEncoding.LINEAR16,
                 encoding= speech.RecognitionConfig.AudioEncoding.ENCODING_UNSPECIFIED,
                 sample_rate_hertz=16000,
                 language_code="en-IN",
                 enable_word_time_offsets=True,
                 enable_automatic_punctuation=True,
             )

             operation = client.long_running_recognize(config=config, audio=audio)

             # Detects speech in the audio file
             response = operation.result(timeout = 500)
             return response
```

   - The transcript obtained is stored as a python dictionary containing the timestamp for each word.



```
In [32]: response=speech_to_text("lecture.mp3")
```

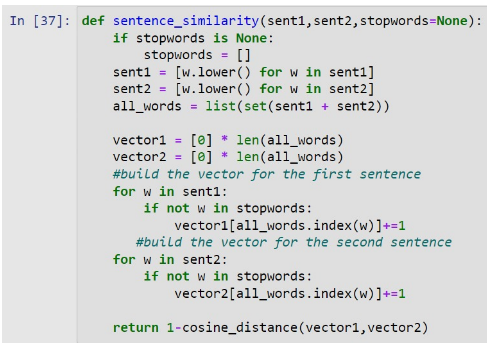
5. Once we get the transcript for the audio, we need to map each sentence to its start and end timestamp. In order to do this, we use the start time of the first word of the sentence as the start time for the sentence and the end time of the last word as the end time of the sentence.



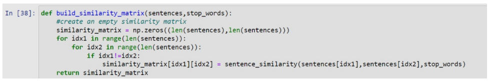
```
In [33]: sentences=[]
         for sentence in response.results:
             sentences.append(sentence)
         transcript=[]
         times=[]
         sentence_map={}
         sentence_arr = []
         for sentence in sentences:
             sentence_transcript=sentence.alternatives[0].transcript+". "
             transcript.append(sentence_transcript)
             time=[]
             start_sec=0
             start_nano=0
             end_sec=0
             end_nano=0
             first_word=sentence.alternatives[0].words[0]
             last_word=sentence.alternatives[0].words[-1]
             start_time=first_word.start_time.seconds
             start_time+=(first_word.start_time.microseconds)/1e6
             end_time=last_word.end_time.seconds
             end_time+=(last_word.end_time.microseconds)/1e6
             time.append(start_time)
             time.append(end_time)
             times.append(time)
             sentence_transcript.replace("[^a-zA-Z0-9]"," ")
             sentence_arr.append(sentence_transcript)
             sentence_map[sentence_transcript]=time
```

6. Now, we can proceed to generation of the text summary. We make use of extractive summarization to accomplish this.

   - First we generate the similarity matrix containing the similarity of each pair of sentences. To calculate the similarity between two sentences, we check the cosine distance between them.

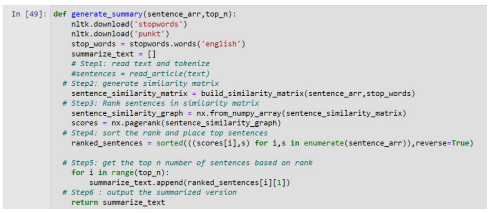


```
In [37]: def sentence_similarity(sent1,sent2,stopwords=None):
             if stopwords is None:
                 stopwords = []
             sent1 = [w.lower() for w in sent1]
             sent2 = [w.lower() for w in sent2]
             all_words = list(set(sent1 + sent2))

             vector1 = [0] * len(all_words)
             vector2 = [0] * len(all_words)
             #build the vector for the first sentence
             for w in sent1:
                 if not w in stopwords:
                     vector1[all_words.index(w)]+=1
                #build the vector for the second sentence
             for w in sent2:
                 if not w in stopwords:
                     vector2[all_words.index(w)]+=1

             return 1-cosine_distance(vector1,vector2)
```

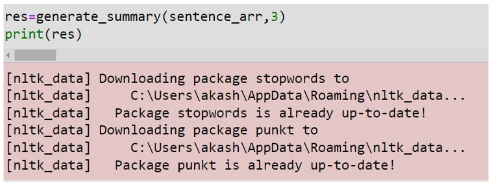


```
In [38]: def build_similarity_matrix(sentences,stop_words):
             #create an empty similarity matrix
             similarity_matrix = np.zeros((len(sentences),len(sentences)))
             for idx1 in range(len(sentences)):
                 for idx2 in range(len(sentences)):
                     if idx1!=idx2:
                         similarity_matrix[idx1][idx2] = sentence_similarity(sentences[idx1],sentences[idx2],stop_words)
             return similarity_matrix
```

- Next step is to rank the sentences in the similarity matrix according to their similarity scores.
- Then we fetch the top n sentences from the sorted list of sentences.
- These top n sentences make up the summarized text.



```
In [49]: def generate_summary(sentence_arr,top_n):
             nltk.download('stopwords')
             nltk.download('punkt')
             stop_words = stopwords.words('english')
             summarize_text = []
             # Step1: read text and tokenize
             #sentences = read_article(text)
           # Step2: generate similarity matrix
             sentence_similarity_matrix = build_similarity_matrix(sentence_arr,stop_words)
           # Step3: Rank sentences in similarity matrix
             sentence_similarity_graph = nx.from_numpy_array(sentence_similarity_matrix)
             scores = nx.pagerank(sentence_similarity_graph)
           # Step4: sort the rank and place top sentences
             ranked_sentences = sorted(((scores[i],s) for i,s in enumerate(sentence_arr)),reverse=True)

           # Step5: get the top n number of sentences based on rank
             for i in range(top_n):
                 summarize_text.append(ranked_sentences[i][1])
           # Step6 : output the summarized version
             return summarize_text
```



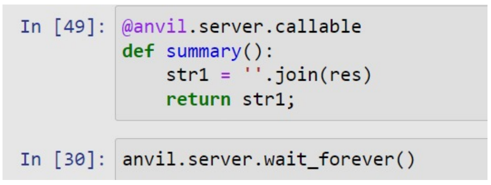
```
res=generate_summary(sentence_arr,3)
print(res)
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\akash\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\akash\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

- This summarized text is then sent to the front-end and displayed to the user as the summarized transcript.

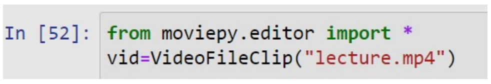
7. Next part if the project involves the video summarization.

    - After fetching the top n sentences of the summary, we make use of the timestamps for the same to generate a timestamp array.

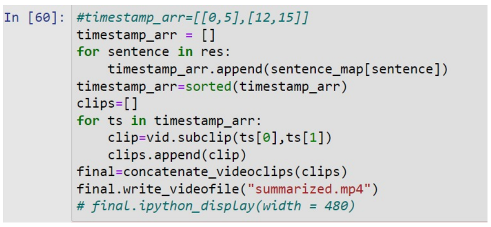


```
In [49]: @anvil.server.callable
         def summary():
             str1 = ''.join(res)
             return str1;

In [30]: anvil.server.wait_forever()
```

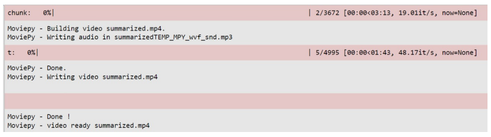
- This is then sorted in ascending order to be used for editing the original video



```
In [52]: from moviepy.editor import *
         vid=VideoFileClip("lecture.mp4")
```

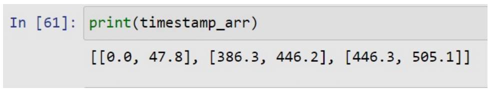


```
In [60]: #timestamp_arr=[[0,5],[12,15]]
         timestamp_arr = []
         for sentence in res:
             timestamp_arr.append(sentence_map[sentence])
         timestamp_arr=sorted(timestamp_arr)
         clips=[]
         for ts in timestamp_arr:
             clip=vid.subclip(ts[0],ts[1])
             clips.append(clip)
         final=concatenate_videoclips(clips)
         final.write_videofile("summarized.mp4")
         # final.ipython_display(width = 480)
```

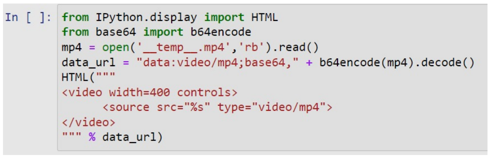


```
chunk:   0%|                                        | 2/3672 [00:00<03:13, 19.01it/s, now=None]

Moviepy - Building video summarized.mp4.
MoviePy - Writing audio in summarizedTEMP_MPY_wvf_snd.mp3
t:   0%|                                            | 5/4995 [00:00<01:43, 48.17it/s, now=None]

MoviePy - Done.
Moviepy - Writing video summarized.mp4


Moviepy - Done !
Moviepy - video ready summarized.mp4
```

- To create the summarized video, we generate clips for each timestamp and then finally concatenate each clip to generate the final summarized video



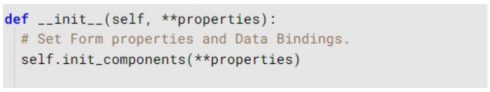
```
In [61]: print(timestamp_arr)

         [[0.0, 47.8], [386.3, 446.2], [446.3, 505.1]]
```



```
In [ ]: from IPython.display import HTML
        from base64 import b64encode
        mp4 = open('__temp__.mp4','rb').read()
        data_url = "data:video/mp4;base64," + b64encode(mp4).decode()
        HTML("""
        <video width=400 controls>
              <source src="%s" type="video/mp4">
        </video>
        """ % data_url)
```

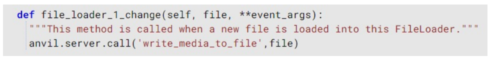
8. Now we have to build the interface with which the user interacts with the system.

    - The first step is to initialize the anvil server at the client side.



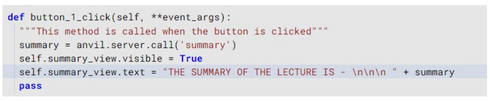
```
def __init__(self, **properties):
  # Set Form properties and Data Bindings.
  self.init_components(**properties)
```

- Next we make use of a file loader where the user can upload the lecture recording to be summarized. This file is then loaded onto the cloud storage as long as processing takes place on it.



```
def file_loader_1_change(self, file, **event_args):
  """This method is called when a new file is loaded into this FileLoader."""
  anvil.server.call('write_media_to_file',file)
```

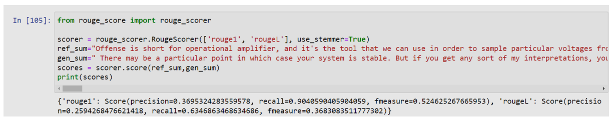
- In order to generate the summary, the button click triggers an event on the back-end which generates the summary and fetches it back to the front end for the user to view.



```
def button_1_click(self, **event_args):
  """This method is called when the button is clicked"""
  summary = anvil.server.call('summary')
  self.summary_view.visible = True
  self.summary_view.text = "THE SUMMARY OF THE LECTURE IS - \n\n\n " + summary
  pass
```

10 RESULTS

We were successfully able to implement a lecture summarizer which is capable of producing a text summary from the transcript of the lecture uploaded by the user in the front-end. The system also produces a summary in the video format. Inorder to evaluate the performance of the system we made use of the ROGUE score of 0.36



```
In [105]: from rouge_score import rouge_scorer

          scorer = rouge_scorer.RougeScorer(['rouge1', 'rougeL'], use_stemmer=True)
          ref_sum="Offense is short for operational amplifier, and it's the tool that we can use in order to sample particular voltages fr
          gen_sum=" There may be a particular point in which case your system is stable. But if you get any sort of my interpretations, yo
          scores = scorer.score(ref_sum,gen_sum)
          print(scores)

          {'rouge1': Score(precision=0.3695324283559578, recall=0.9040590405904059, fmeasure=0.524625267665953), 'rougeL': Score(precisio
          n=0.2594268476621418, recall=0.6346863468634686, fmeasure=0.3683083511777302)}
```

11 CONCLUSION

We were successfully able to build a tool that is capable of summarizing a video lecture by providing a text summary and a summarized video corresponding to the summarized text. The system faces difficulty in transcribing audio of spontaneous speech, this is a downside of the text-to-speech conversion that we are making use of. We were able to achieve ROUGE scores of 0.36

12 FUTURE WORK

As discussed, the system faces difficulty in transcribing audio of spontaneous speech. Since the transcription part is an important part of generating the summary, it affects the results obtained. A further improvement to the project would be to make the system more robust with respect to spontaneous speech. The system also gives poor results in case of different accents in which the audio is recorded. This can be another improvement to the system that can be included.

13 REFERENCES

1. https://www.researchgate.net/publication/228553277_Intelligent_Transcription_System_Based_on_Spontaneous_Speech_Processing

2. https://ieeexplore.ieee.org/abstract/document/1306513

3. https://dl.acm.org/doi/abs/10.1145/3232676

4. https://www.jstor.org/stable/jeductechsoci.17.4.65

5. https://ieeexplore.ieee.org/document/4161209

6. https://arxiv.org/abs/1612.01744

7. https://dergipark.org.tr/en/pub/pap/article/371658

8. https://www.researchgate.net/publication/324818574_Automatic_Speech_Recognition_-_Spontaneous_Speech

9. https://ieeexplore.ieee.org/abstract/document/6754792/

10. https://ieeexplore.ieee.org/abstract/document/7944061

11. https://link.springer.com/chapter/10.1007/978-3-319-10599-4_35

12. https://dl.acm.org/doi/pdf/10.1145/290747.290773

13. https://link.springer.com/chapter/10.1007/978-3-642-33564-8_1

14. https://proceedings.neurips.cc/paper/2014/hash/0eec27c419d0fe24e53c90338cdc8bc6-Abstract.html

15. https://scholar.google.com/scholar?start=40&q=video+summarization&hl=en&as_sdt=0,5

16. https://dl.acm.org/doi/abs/10.1145/3123266.3123328

17. https://link.springer.com/article/10.1007/s11263-014-0794-5

18. https://www.mdpi.com/2076-3417/8/8/1397