A Project Report

On

PODCAST SUMMARIZATION MODEL

BY

Harshveer Singh Thind (SE21UARI047)

Srija Reddy K (SE21UARI158)

A.Siddharth Reddy (SE21UARI020)

TanmayiSri V (SE21UARI167)

Janvi Agarwal (SE21UARI055)

SUBMITTED IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS OF

COURSE PROJECT IN AI 3106: Foundations of NLP

ECOLE CENTRALE SCHOOL OF ENGINEERING

HYDERABAD

(December 2023)

PROBLEM STATEMENT

The problem statement we have chosen is building a podcast summarizer using speech to text model and a summarizing model. Initially while deciding our problem statement, we wanted to work on something that has not been developed and is beneficial to the community. There are a few podcast summarizers available online, but they are not completely developed and hence we thought it could be an area that could be explored and worked upon. There are techniques available on how to create a model, and after extensive research we concluded that using Speech recognition APIs and hugging face model would be the best fit. Our system aims to condense lengthy podcast episodes (our dataset contains podcasts that are 1 hour long) into shorter, catchy summaries. Podcasts today have gained wide attention due to its diverse range of topics and interests aligning to the user's needs. Listeners have the flexibility to consume the content at their convenience, but lengthy podcasts could be monotonous and tedious. The subscribers usually prefer short, easy-to-understand podcasts and our model is the perfect solution for it. Concise summaries can also be helpful as people can extract key takeaways without having to listen to the entire episode.

Contributions of each individual:

- Janvi worked on the documentation, data scrapping and extracting audio file from a data source.
- Tanmayi Sri and Harshveer worked on converting the input audio file (speech) to text using Google web speech APIs through the speech recognition library.
- Siddharth and Srija worked on summarization model using BART model and the hugging face transformer.
- Srija also worked on evaluating the accuracy of the model by utilising rouge package in python to assess the quality of a system generated summary by comparing it to a reference summary.

CONTENTS

Title page……..…… 1

Problem Definition……..... 2

1. Implementation ……..…….....…..... 4

2. Results and Discussion ……..... 5

References…….....XX

Appendix A…….....XX

Appendix B…….....XX

IMPLEMENTATION

Data sets used:

Since we worked on a pre-trained model we did not require any datasets. Our code does not require a dataset, we used a pre- trained model .
We used an audio file from the 'The popular Political dataset' and converted the speech to text.
To perform summarization, we essentially used BART (Bidirectional and Auto-Regressive Transformers) pre-trained model for abstractive text summarization available in the Hugging Face Transformers library.

Environment:

Google colab, Notepad++, Jupyter Notebook

Proposed Technique / Methodology:

LINE BY LINE BREAKDOWN OF OUR CODE PODCAST SEGMENTATION

AUDIO TO TEXT MODEL:

```
import speech_recognition as sr

def audio_to_text(audio_file_path):
    # Initialize the recognizer
    recognizer = sr.Recognizer()

    # Load the audio file
    with sr.AudioFile(audio_file_path) as source:
        # Adjust for ambient noise
        recognizer.adjust_for_ambient_noise(source)

        # Record the audio
        audio = recognizer.record(source)

    try:
        # Recognize speech using Google Web Speech API
        text = recognizer.recognize_google(audio)
        return text
    except sr.UnknownValueError:
        print("Google Web Speech API could not understand audio")
    except sr.RequestError as e:
        print(f"Could not request results from Google Web Speech API; {e}")
```

This code block defines a function called audio_to_text that utilizes the SpeechRecognition library to convert speech from an audio file to text. The function takes the path to an audio file as its input. Inside the function, a speech recognizer is initialized, and the ambient noise in the audio file is adjusted for better recognition. The audio is then recorded from the specified file, and an attempt is made to recognize the speech using the Google Web Speech API. If successful, the recognized text is returned. If the API fails to understand the audio or encounters an error during the request appropriate error messages are printed.

SEGMENTING PODCAST

Our code uses the **Pydub** library to split a large audio file into smaller segments.

Here is a detailed explanation of our script:

1.First, we import the libraries, Pydub and os

- These lines import the necessary libraries. **Pydub** is used for audio processing, and the **os** module is used for interacting with the operating system.

2.We define the split_audio function :

```
def split_audio(input_file, output_folder, segment_duration=15):
```

- This function takes three parameters:
  Input_file: The path to the large audio file that you want to split.
  Output_folder: The folder where the smaller audio segments will be saved.
  Segment_duration: The duration (in seconds) of each audio segment. The default is set to 15 seconds.

3.Creation of the output folder.

```
if not os.path.exists(output_folder):
        os.makedirs(output_folder)
```

- create an output folder.

4.Loading the audio file.

```
audio = AudioSegment.from_wav(input_file)
```

load the large audio file.

5.Looping through the audio file and creating segments.

```
for start_time in range(0, len(audio), segment_duration * 1000):
        end_time = min(start_time + segment_duration * 1000, len(audio)) segment =
        audio[start_time:end_time]

segment_path = os.path.join(output_folder, f"segment_{start_time//1000}- {end_time//1000}.wav")

segment.export(segment_path, format="wav")
```

- This loop iterates through the audio file. It extracts a segment of audio from **start_time** to **end_time** for each iteration and then exports the segment as a WAV file to the specified output folder.

6. Specifying the Input and Output paths and calling the function.

TEXT TO SPEECH FOR ENTIRE PODCAST

1. We import the **'Path'** class from the **'pathlib'** module.

2.input_dir = Path("/content/audio_segment"):
This will be the directory where the script looks for audio files.

3.files = list(input_dir.rglob("*.wav*")):
Use the '**rglob'**method to recursively search for files with the extension **".wav"**

4.Call a function '**audio_to_text'** with the path to the current audio file as an argument. This function is converts the audio content into text.

5.if text is not None:

- Check if the '**text'** variable contains a non-None value, meaning that the speech recognition process was successful. This is done because an audio segment might contain no dialogues which may result in an error.

In summary, this segment of our code reads audio files with the ".wav" extension from the specified directory, applies a speech recognition function (`audio_to_text`) to convert the audio content into text, and writes the recognized text to a text file named "mytxt.txt". If speech recognition fails for any file, an error message is printed to the console.

1.Extract timestamps from filenames
timestamps = [int(str(path.stem).split('_')[-1].split('-')[0]) for path in file_paths]

- Here, a list named '**timestamps'** is created using a list comprehension. It extracts the start timestamp from each file's name by splitting the filename. It converts the extracted timestamp to an integer.

2.Combine file paths and timestamps, then sort based on timestamps
sorted_files = sorted(zip(file_paths, timestamps), key=lambda x: x[1])

- This line combines the file paths and timestamps into tuples using the '**zip'** function. The resulting list of tuples is then sorted based on the timestamps using the '**sorted'** function. The **'key'** argument specifies a lambda function that extracts the timestamp from each tuple.

3.Extract only file paths from the sorted list
sorted_file_paths = [path for path, timestamp in sorted_files]

- In this part, a list named '**sorted_file_paths'** is created using a list comprehension. It extracts only the file paths from the sorted list of tuples.

4.textfile = open('/content/Start-.txt', 'w'):

- This line opens a text file named **"Start-.txt"** in write mode **('w').** If the file doesn't exist, it will be created. If it already exists, its previous content will be overwritten.

5.text = audio_to_text(str(sorted_file_paths[i])):

- For each file path, it calls the function '**audio_to_text()'** with the current file path as an argument. The function seems to convert audio data from the file into text. The result is stored in the variable '**text'.**

So, in summary, this code reads audio files from a list of file paths (`sorted_file_paths`), converts the audio content to text using the `audio_to_text` function, and writes the resulting text to a file named "Start-.txt". If the text extraction fails for any file, it prints an error message.

SUMMARIZATION MODEL

This script does the following:

1. Load Text File into DataFrame:

It reads the content of a text file **('Start- (1).txt')** and stores it in the variable **'text_content'**. The content is then used to create a pandas **DataFrame ('df')** with a single column named **'article content'.**

```
txt_file_path = "/content/Start- (1).txt"
with open(txt_file_path, 'r', encoding='utf-8') as file:
text_content = file.read()
df = pd.DataFrame({'article content': [text_content]})
```

2.Create Summarization Pipeline

It uses the Hugging Face Transformers library to create a summarization pipeline. The chosen model is '**sshleifer/distilbart-cnn-12-6**'.

```
summarizer = pipeline("summarization", model="sshleifer/distilbart-cnn-12-6")
```

3.Generate Summary Function

The script defines a function '**generate_summary'** that takes an article as input.
It splits the article into chunks of 1000 tokens and generates a summary for each chunk. The summaries for each chunk are then combined into a single summary.

4.Apply Summarization Function to DataFrame

The **'generate_summary'** function is applied to the **'article content'** column of the **DataFrame** using the **`apply`** method, and the results are stored in a new column called **'Summary'.**

```
df['Summary'] = df['article content'].apply(generate_summary)
```

5.Set Pandas Options

It sets pandas options to display the entire content of the **'Summary'** column.

```
pd.set_option('display.max_colwidth', None)
```

6.Print or Save DataFrame with Summary

It prints the DataFrame containing the summary column.

```
print(df[['Summary']])
```

7.Save DataFrame to New Text File

It saves the **DataFrame** with the summary to a new text file **('Start- (1)_summary.txt')** using the **`to_csv`** method.

```
df[['Summary']].to_csv("/content/Start- (1)_summary.txt", index=False, header=False, sep='\t')
```

This script essentially reads a text file, summarizes its content using a pre-trained model, and then saves the original and summarized content to a new text file.

RESULTS AND DISCUSSION

```
from rouge import Rouge

# Read the reference summary from a file
reference_file_path = "/Users/srijareddy/Desktop/hi/reference_summary.txt"  # Replace with the path to your reference
with open(reference_file_path, 'r', encoding='utf-8') as file:
    reference_summary = file.read()

# Read the system summary from a file
system_file_path = "/Users/srijareddy/Desktop/hi/summary1.txt"  # Replace with the path to your system summary file
with open(system_file_path, 'r', encoding='utf-8') as file:
    system_summary = file.read()

# Initialize ROUGE
rouge = Rouge()

# Calculate ROUGE scores
scores = rouge.get_scores(system_summary, reference_summary)

# Print the ROUGE scores
print(scores)
```

For the evaluation part, we utilized the ROUGE (Recall-Oriented Understudy for Gisting Evaluation) package to collectively assess the quality of a system-generated summary by comparing it to a reference summary. Here's a breakdown of our group effort:

Importing the ROUGE Module:

We began by importing the Rouge class from the ROUGE package. This class provided the essential functionality needed to calculate ROUGE scores for our evaluation.

Reading the Reference Summary:

We read the reference summary from a specified file path (reference_file_path). This served as the human-generated or gold-standard reference against which the system summary would be collectively evaluated.

Reading the System Summary:

Similarly, we read the system-generated summary from a specified file path (system_file_path). This summary represented the output produced by the system or algorithm that we aimed to evaluate.

Initializing ROUGE:

We created an instance of the Rouge class, allowing us to collectively utilize the ROUGE metric to assess the quality of the system-generated summary in comparison to the reference summary.

Calculating ROUGE Scores:

Utilizing the **get_scores** method of the Rouge class, we collectively calculated ROUGE scores by comparing the system-generated summary to the reference summary. The resulting scores variable was a dictionary containing various ROUGE metrics, including precision, recall, and F1 scores.

Printing the ROUGE Scores:

We then printed the calculated ROUGE scores. This step provided our team with a comprehensive view of how well the system-generated summary aligned with the reference summary.

In conclusion, our collaborative effort in writing this code facilitated the evaluation of a system-generated summary's performance. Leveraging the ROUGE metric, a widely used measure in natural language processing and summarization tasks, our group gained valuable insights into precision, recall, and F1 scores. This collective assessment aided in evaluating the overall quality of the summary.

Output:

```
[{'rouge-1': {'r': 0.2553191489361702, 'p': 0.19933554817275748, 'f': 0.22388059209073577}, 'rouge-2': {'r':
0.05542168674698795, 'p': 0.0465587044534413, 'f': 0.050605055543816696}, 'rouge-l': {'r': 0.2425531914893617, 'p':
0.1893687707641196, 'f': 0.2126856223998952}}]
```

The output you provided is a dictionary containing ROUGE scores for various metrics, specifically for unigrams (ROUGE-1), bigrams (ROUGE-2), and the longest common subsequence (ROUGE-L).

- The ROUGE-1 scores suggest a moderate level of overlap between unigrams in the system-generated and reference summaries.
- The ROUGE-2 scores indicate a lower level of agreement in capturing bigrams, suggesting room for improvement.
- The ROUGE-L scores align with ROUGE-1, emphasizing the importance of common subsequences.

In conclusion, while the system-generated summary demonstrated reasonable performance for unigrams, improvements are needed, especially concerning bigrams. These metrics provide valuable insights into the effectiveness of the summarization algorithm, helping guide refinements for enhanced performance.

APPENDIX A

URL TO COLAB PYTHON SCRIPT:

here you go!

LINK TO GITHUB REPO:

Click here to see magic

APPENDIX B

NLP_Project.ipynb - Colaboratory.pdf