Task 1: Semantic Chunking of a Youtube Video

Problem Statement:

The objective is to extract high-quality, meaningful (semantic) segments from a specified YouTube video.

Suggested workflow:

1. **Download Video and Extract Audio:** Download the video and separate the audio component.
2. **Transcription of Audio:** Utilize an open-source Speech-to-Text model to transcribe the audio. *Provide an explanation of the chosen model and any techniques used to enhance the quality of the transcription.*
3. **Time-Align Transcript with Audio:** *Describe the methodology and steps for aligning the transcript with the audio.*
4. **Semantic Chunking of Data:** Slice the data into audio-text pairs, using both semantic information from the text and voice activity information from the audio, with each audio-chunk being less than 15s in length. *Explain the logic used for semantic chunking and discuss the strengths and weaknesses of your approach.*

- **Output Format:** Provide the results as a list of dictionaries, each representing a semantic chunk. Each dictionary should include:
    - `chunk_id` : A unique identifier for the chunk (integer).
    - `chunk_length` : The duration of the chunk in seconds (float).
    - `text` : The transcribed text of the chunk (string).
    - `start_time` : The start time of the chunk within the video (float).
    - `end_time` : The end time of the chunk within the video (float).

```
sample_output_list = [
    {
        "chunk_id": 1,
        "chunk_length": 14.5,
        "text": "Here is an example of a semantic chunk from the video.",
        "start_time": 0.0,
        "end_time": 14.5,
    },
    # Additional chunks follow...
]
```

Setup Whisper & Other Libraries

This might restart the kernel once or twice just restart it and proceed further

Show code

Show hidden output

```
# Helper functions for the video and audio extraction
from pytube import YouTube
from moviepy.editor import VideoFileClip

def download_video(youtube_url, save_path="./"):
    try:
        # Create a YouTube object with the link
        yt = YouTube(youtube_url)

        # Get the highest resolution stream
        stream = yt.streams.get_highest_resolution()

        # Generate a unique filename for the downloaded video
        video_filename = "video.mp4"

        # Download the video
        video_path = stream.download(output_path=save_path, filename=video_filename)
        print("Download complete!")

        # Extract audio
        audio_path = extract_audio(video_path, save_path)
        if audio_path:
            print("Audio extracted successfully at:", audio_path)
        return [video_path, audio_path]
    except Exception as e:
        print("An error occurred:", str(e))


def extract_audio(video_path, save_path):
    try:
        # Load the video clip using moviepy
        video_clip = VideoFileClip(video_path)

        # Extract audio
        audio_clip = video_clip.audio

        # Create path for saving audio
        audio_path = video_path[:-4] + ".mp3"

        # Save audio
        audio_clip.write_audiofile(audio_path)

        return audio_path

    except Exception as e:
        print("An error occurred while extracting audio:", str(e))

# # Example usage
# youtube_link = input("Enter the YouTube link: ")
# download_video(youtube_link)
```

```
#@markdown **Run Whisper and Perform the Audio Chunking and g

# @markdown Required settings:
youtube_link = "https://www.youtube.com/watch?v=Sby1uJ_NFIY"
audio_path = download_video(youtube_link)[1]
model_size = "medium"  # @param ["medium", "large"]
language = "english"  # @param {type:"string"}
translation_mode = "End-to-end Whisper (default)"  # @param [
# @markdown Advanced settings:
deepl_authkey = ""
source_separation = False  # @param {type:"boolean"}
vad_threshold = 0.4  # @param {type:"number"}
chunk_threshold = 3.0  # @param {type:"number"}
deepl_target_lang = "EN-US"
max_attempts = 1  # @param {type:"integer"}
initial_prompt = ""

import tensorflow as tf
import torch
import whisper
import os
import ffmpeg
import srt
from tqdm import tqdm
import datetime
import deepl
import urllib.request
import json
from google.colab import files
```

Run Whisper and Perform the Audio Chunking and generate .srt file

Required settings:

youtube_link: " https://www.youtube.com/watch?v=Sby "

model_size: medium

language: " english "

translation_mode: End-to-end Whisper (default)

Advanced settings:

source_separation: ☐

vad_threshold: 0.4

chunk_threshold: 3.0

max_attempts: 1

```
# Configuration
assert max_attempts >= 1
assert vad_threshold >= 0.01
assert chunk_threshold >= 0.1
assert audio_path != ""
assert language != ""
if translation_mode == "End-to-end Whisper (default)":
    task = "translate"
    run_deepl = False
elif translation_mode == "Whisper -> DeepL":
    task = "transcribe"
    run_deepl = True
elif translation_mode == "No translation":
    task = "transcribe"
    run_deepl = False
else:
    raise ValueError("Invalid translation mode")
if initial_prompt.strip() == "":
    initial_prompt = None

if "http://" in audio_path or "https://" in audio_path:
    print("Downloading audio...")
    urllib.request.urlretrieve(audio_path, "input_file")
    audio_path = "input_file"
else:
    if not os.path.exists(audio_path):
        try:
            audio_path = uploaded_file
            if not os.path.exists(audio_path):
                raise ValueError("Input audio not found. Is y
        except NameError:
            raise ValueError("Input audio not found. Did you

out_path = os.path.splitext(audio_path)[0] + ".srt"
out_path_pre = os.path.splitext(audio_path)[0] + "_Untranslat
if source_separation:
    print("Separating vocals...")
    !ffprobe -i "{audio_path}" -show_entries format=duration
    with open("input_length") as f:
        input_length = int(float(f.read())) + 1
    !spleeter separate -d {input_length} -p spleeter:2stems -
    spleeter_dir = os.path.basename(os.path.splitext(audio_pa
    audio_path = "output/" + spleeter_dir + "/vocals.wav"

print("Encoding audio...")
if not os.path.exists("vad_chunks"):
    os.mkdir("vad_chunks")
ffmpeg.input(audio_path).output(
    "vad_chunks/silero_temp.wav",
    ar="16000",
    ac="1",
    acodec="pcm_s16le",
    map_metadata="-1",
    fflags="+bitexact",
).overwrite_output().run(quiet=True)

print("Running VAD...")
model, utils = torch.hub.load(
    repo_or_dir="snakers4/silero-vad", model="silero_vad", or
)

(get_speech_timestamps, save_audio, read_audio, VADIterator,

# Generate VAD timestamps
VAD_SR = 16000
wav = read_audio("vad_chunks/silero_temp.wav", sampling_rate=
t = get_speech_timestamps(wav, model, sampling_rate=VAD_SR, t

# Add a bit of padding, and remove small gaps
for i in range(len(t)):
    t[i]["start"] = max(0, t[i]["start"] - 3200)  # 0.2s head
    t[i]["end"] = min(wav.shape[0] - 16, t[i]["end"] + 20800)
    if i > 0 and t[i]["start"] < t[i - 1]["end"]:
        t[i]["start"] = t[i - 1]["end"]  # Remove overlap

# If breaks are longer than chunk_threshold seconds, split in
# This'll effectively turn long transcriptions into many shor
u = [[]]
for i in range(len(t)):
    if i > 0 and t[i]["start"] > t[i - 1]["end"] + (chunk_thr
        u.append([])
    u[-1].append(t[i])
```

```
# Merge speech chunks
for i in range(len(u)):
    save_audio(
        "vad_chunks/" + str(i) + ".wav",
        collect_chunks(u[i], wav),
        sampling_rate=VAD_SR,
    )
os.remove("vad_chunks/silero_temp.wav")

# Convert timestamps to seconds
for i in range(len(u)):
    time = 0.0
    offset = 0.0
    for j in range(len(u[i])):
        u[i][j]["start"] /= VAD_SR
        u[i][j]["end"] /= VAD_SR
        u[i][j]["chunk_start"] = time
        time += u[i][j]["end"] - u[i][j]["start"]
        u[i][j]["chunk_end"] = time
        if j == 0:
            offset += u[i][j]["start"]
        else:
            offset += u[i][j]["start"] - u[i][j - 1]["end"]
        u[i][j]["offset"] = offset

# Run Whisper on each audio chunk
print("Running Whisper...")
model = whisper.load_model(model_size)
subs = []
segment_info = []
sub_index = 1
suppress_low = [
    "Thank you",
    "Thanks for",
    "ike and ",
    "Bye.",
    "Bye!",
    "Bye bye!",
    "lease sub",
    "The end.",
    "視聴",
]
suppress_high = [
    "ubscribe",
    "my channel",
    "the channel",
    "our channel",
    "ollow me on",
    "for watching",
    "hank you for watching",
    "for your viewing",
    "r viewing",
    "Amara",
    "next video",
    "full video",
    "ranslation by",
    "ranslated by",
    "ee you next week",
    "ご視聴",
    "視聴ありがとうございました",
]
for i in tqdm(range(len(u))):
    line_buffer = []  # Used for DeepL
    for x in range(max_attempts):
        result = model.transcribe(
            "vad_chunks/" + str(i) + ".wav", task=task, langu
        )
        # Break if result doesn't end with severe hallucinati
        if len(result["segments"]) == 0:
            break
        elif result["segments"][-1]["end"] < u[i][-1]["chunk_
            break
        elif x+1 < max_attempts:
            print("Retrying chunk", i)
    for r in result["segments"]:
        # Skip audio timestamped after the chunk has ended
        if r["start"] > u[i][-1]["chunk_end"]:
            continue
        # Reduce log probability for certain words/phrases
        for s in suppress_low:
            if s in r["text"]:
                r["avg_logprob"] -= 0.15
        for s in suppress_high:
```

```
        for s in suppress_high:
            if s in r["text"]:
                r["avg_logprob"] -= 0.35
        # Keep segment info for debugging
        del r["tokens"]
        segment_info.append(r)
        # Skip if log prob is low or no speech prob is high
        if r["avg_logprob"] < -1.0 or r["no_speech_prob"] > 0
            continue
        # Set start timestamp
        start = r["start"] + u[i][0]["offset"]
        for j in range(len(u[i])):
            if (
                r["start"] >= u[i][j]["chunk_start"]
                and r["start"] <= u[i][j]["chunk_end"]
            ):
                start = r["start"] + u[i][j]["offset"]
                break
        # Prevent overlapping subs
        if len(subs) > 0:
            last_end = datetime.timedelta.total_seconds(subs[
            if last_end > start:
                subs[-1].end = datetime.timedelta(seconds=sta
        # Set end timestamp
        end = u[i][-1]["end"] + 0.5
        for j in range(len(u[i])):
            if r["end"] >= u[i][j]["chunk_start"] and r["end"
                end = r["end"] + u[i][j]["offset"]
                break
        # Add to SRT list
        subs.append(
            srt.Subtitle(
                index=sub_index,
                start=datetime.timedelta(seconds=start),
                end=datetime.timedelta(seconds=end),
                content=r["text"].strip(),
            )
        )
        sub_index += 1

with open("segment_info.json", "w", encoding="utf8") as f:
    json.dump(segment_info, f, indent=4)

# DeepL translation
translate_error = False
if run_deepl:
    print("Translating...")
    with open(out_path_pre, "w", encoding="utf8") as f:
        f.write(srt.compose(subs))
    print("(Untranslated subs saved to", out_path_pre, ")")

    lines = []
    punct_match = ["。", "、", "，", "．", "～", "！", "!", "？"
    for i in range(len(subs)):
        if language.lower() == "japanese":
            if subs[i].content[-1] not in punct_match:
                subs[i].content += "。"
            subs[i].content = "「" + subs[i].content + "」"
        else:
            if subs[i].content[-1] not in punct_match:
                subs[i].content += "."
            subs[i].content = '"' + subs[i].content + '"'
    for i in range(len(subs)):
        lines.append(subs[i].content)

    grouped_lines = []
    english_lines = []
    for i, l in enumerate(lines):
        if i % 30 == 0:
            # Split lines into smaller groups, to prevent err
            grouped_lines.append([])
            if i != 0:
                # Include previous 3 lines, to preserve conte
                grouped_lines[-1].extend(grouped_lines[-2][-3
        grouped_lines[-1].append(l.strip())

    try:
        translator = deepl.Translator(deepl_authkey)
        for i, n in enumerate(tqdm(grouped_lines)):
            x = ["\n".join(n).strip()]
            if language.lower() == "japanese":
                result = translator.translate_text(x, source_
            else:
```

```
                result = translator.translate_text(x, target_
            english_tl = result[0].text.strip().splitlines()
            assert len(english_tl) == len(n), (
                "Invalid translation line count ("
                + str(len(english_tl))
                + " vs "
                + str(len(n))
                + ")"
            )
            if i != 0:
                english_tl = english_tl[3:]
            remove_quotes = dict.fromkeys(map(ord, '"„“‟" 「
            for e in english_tl:
                english_lines.append(
                    e.strip().translate(remove_quotes).replac
                )
        for i, e in enumerate(english_lines):
            subs[i].content = e
    except Exception as e:
        print("DeepL translation error:", e)
        print("(downloading untranslated version instead)")
        translate_error = True

# Write SRT file
if translate_error:
    files.download(out_path_pre)
else:
    # Removal of garbage lines
    garbage_list = [
        "a",
        "aa",
        "ah",
        "ahh",
        "ha",
        "haa",
        "hah",
        "haha",
        "hahaha",
        "mmm",
        "mm",
        "m",
        "h",
        "o",
        "mh",
        "mmh",
        "hm",
        "hmm",
        "huh",
        "oh",
    ]
    need_context_lines = [
        "feelsgod",
        "godbye",
        "godnight",
        "thankyou",
    ]
    clean_subs = list()
    last_line_garbage = False
    for i in range(len(subs)):
        c = subs[i].content
        c = (
            c.replace(".", "")
            .replace(",", "")
            .replace(":", "")
            .replace(";", "")
            .replace("!", "")
            .replace("?", "")
            .replace("–", " ")
            .replace("  ", " ")
            .replace("  ", " ")
            .replace("  ", " ")
            .lower()
            .replace("that feels", "feels")
            .replace("it feels", "feels")
            .replace("feels good", "feelsgood")
            .replace("good bye", "goodbye")
            .replace("good night", "goodnight")
            .replace("thank you", "thankyou")
            .replace("aaaaaa", "a")
            .replace("aaaa", "a")
            .replace("aa", "a")
            .replace("aa", "a")
            .replace("mmmmmm", "m")
```

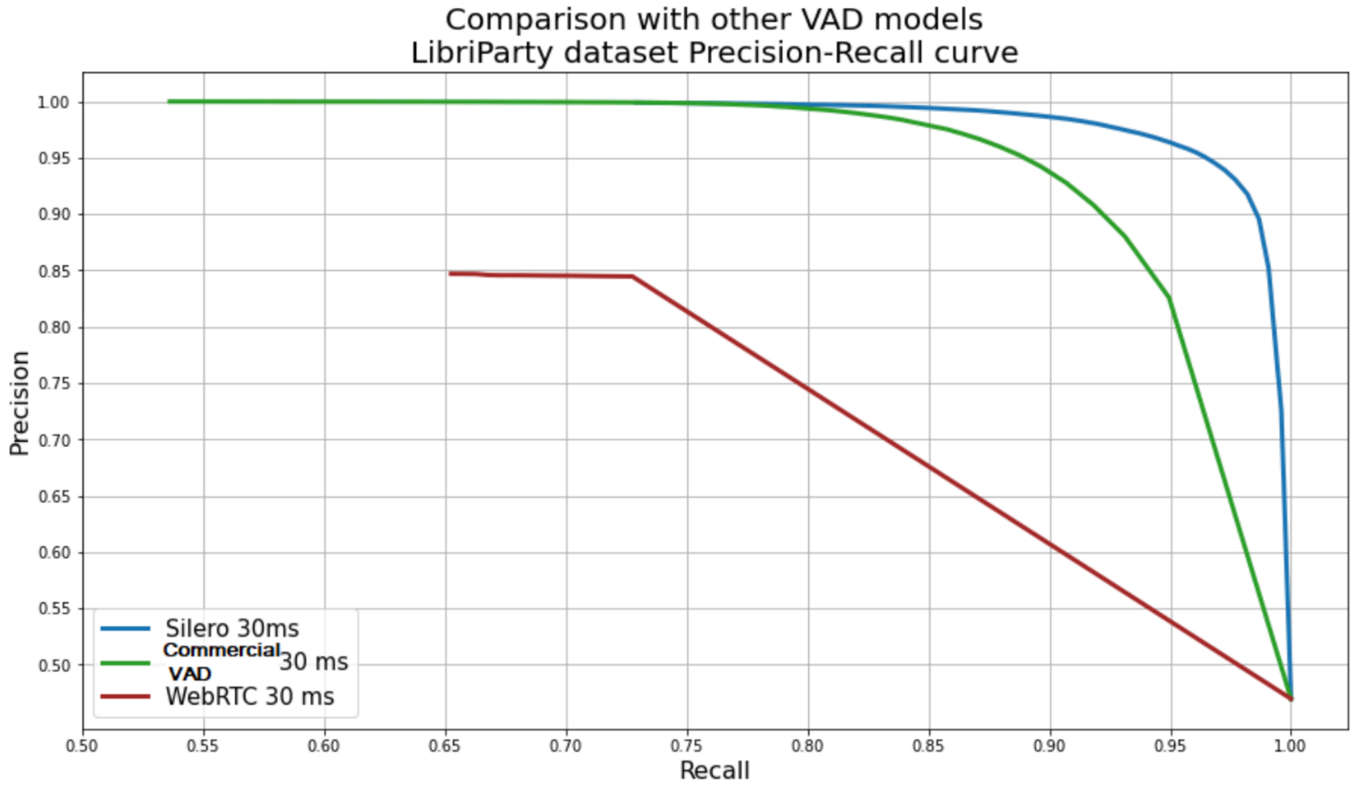
```
            .replace("mmmm", "m")
            .replace("mm", "m")
            .replace("mm", "m")
            .replace("hhhhhh", "h")
            .replace("hhhh", "h")
            .replace("hh", "h")
            .replace("hh", "h")
            .replace("oooooo", "o")
            .replace("oooo", "o")
            .replace("oo", "o")
            .replace("oo", "o")
        )
        is_garbage = True
        for w in c.split(" "):
            if w.strip() == "":
                continue
            if w.strip() in garbage_list:
                continue
            elif w.strip() in need_context_lines and last_lir
                continue
            else:
                is_garbage = False
                break
        if not is_garbage:
            clean_subs.append(subs[i])
        last_line_garbage = is_garbage
    with open(out_path, "w", encoding="utf8") as f:
        f.write(srt.compose(clean_subs))
    print("\nDone! Subs written to", out_path)
    print("Downloading SRT file:")
    files.download(out_path)
```

```
Download complete!
MoviePy - Writing audio in /content/./video.mp3
MoviePy - Done.
Audio extracted successfully at: /content/./video.mp3
Encoding audio...
Running VAD...
WARNING:py.warnings:/usr/local/lib/python3.10/dist-packages/torch/hub.py:294: UserWarning: You are about to download and
  warnings.warn(

Downloading: "https://github.com/snakers4/silero-vad/zipball/master" to /root/.cache/torch/hub/master.zip
Running Whisper...
100%|████████████████████████████████████████| 1.42G/1.42G [00:14<00:00, 108MiB/s]
  0%|          | 0/2 [00:00<?, ?it/s]WARNING:py.warnings:/usr/local/lib/python3.10/dist-packages/whisper/transcribe.py:1
  warnings.warn("FP16 is not supported on CPU; using FP32 instead")

  0%|          | 0/2 [27:33<?, ?it/s]
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-3-4eaa949bf1dc> in <cell line: 175>()
    176     line_buffer = []  # Used for DeepL
    177     for x in range(max_attempts):
--> 178         result = model.transcribe(
    179             "vad_chunks/" + str(i) + ".wav", task=task, language=language, initial_prompt=initial_prompt
    180         )

                              17 frames
/usr/local/lib/python3.10/dist-packages/whisper/model.py in qkv_attention(self, q, k, v, mask)
    106
    107         w = F.softmax(qk, dim=-1).to(q.dtype)
--> 108         return (w @ v).permute(0, 2, 1, 3).flatten(start_dim=2), qk.detach()
    109
    110

KeyboardInterrupt:
```

Check the json file in the folder to see the output here is the link to an already generated file [segment info json file]

Models & Libraries used and Why

For Chunking I have used Silero VAD which has a high precision score and is faster than WebRTC and commercial VAD and useful for real-time applications.



Metrics

ROC-AUC score

> SpeechBrain VAD does not support streaming, i.e. it looks at the entire audio context, so we didn't include it in the Precision-Recall curves.

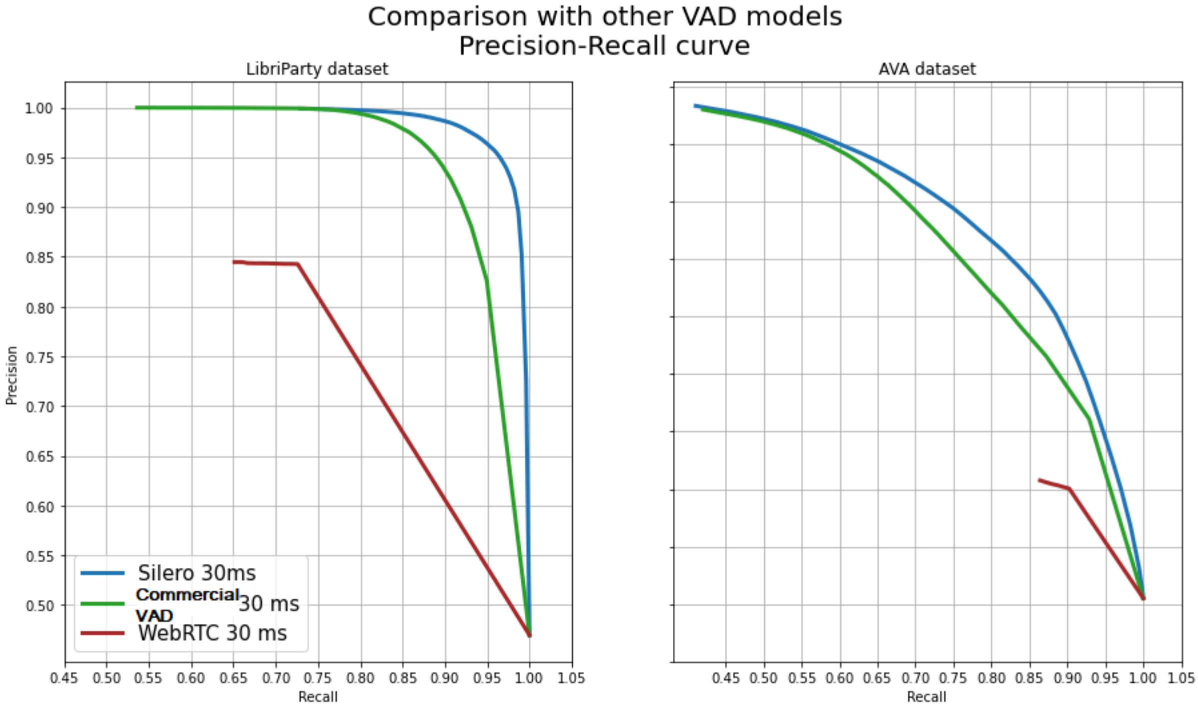
Model	AVA	LibryParty	Streaming
Silero v4 (current) 16k	0.9	0.99	✔
Silero v4 (current) 8k	0.89	0.97	✔
Silero v3 16k	0.87	0.93	✔
Silero v3 8k	0.85	0.94	✔
SpeechBrain	0.85	0.99	❌
WebRTC	0.66	0.81	✔
Unnamed commercial VAD	0.88	0.97	✔

Silero VAD vs Other Available Solutions

Parameters: `16000 Hz` sampling rate, `30 ms` (512 samples).

WebRTC VAD algorithm is extremely fast and pretty good at separating noise from silence, but pretty poor at separating speech from noise.

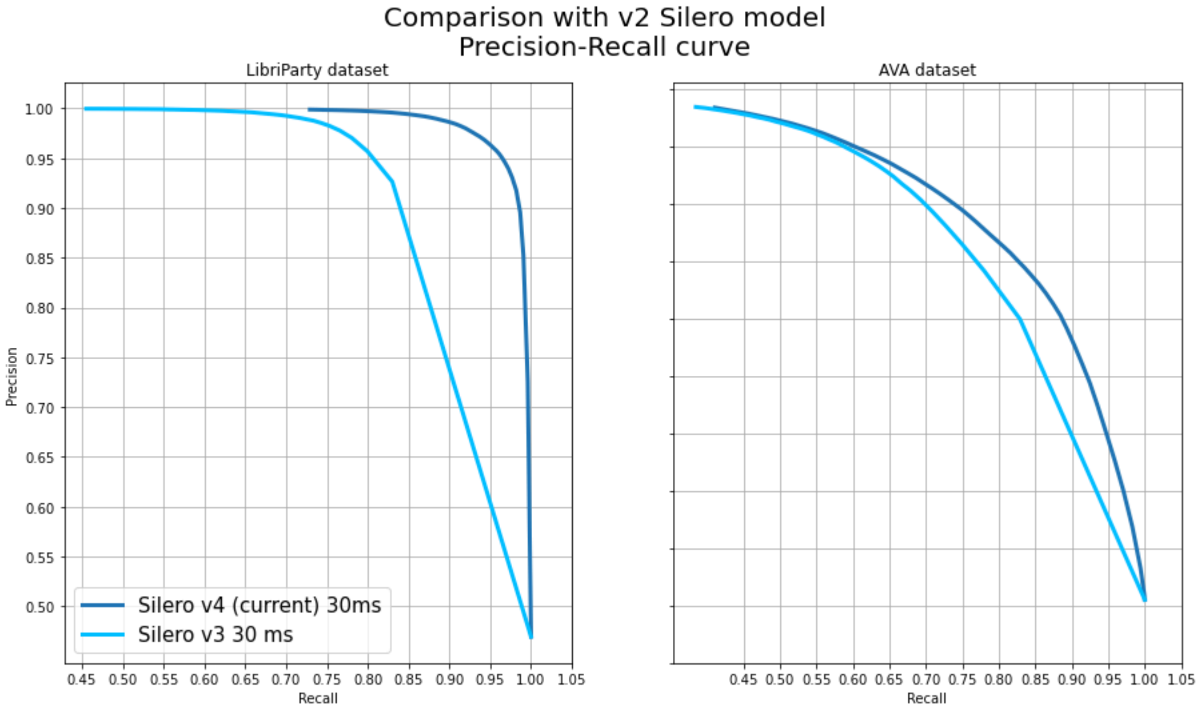
Picovoice VAD is good overall, but we were able to surpass it in quality (eof 2022).



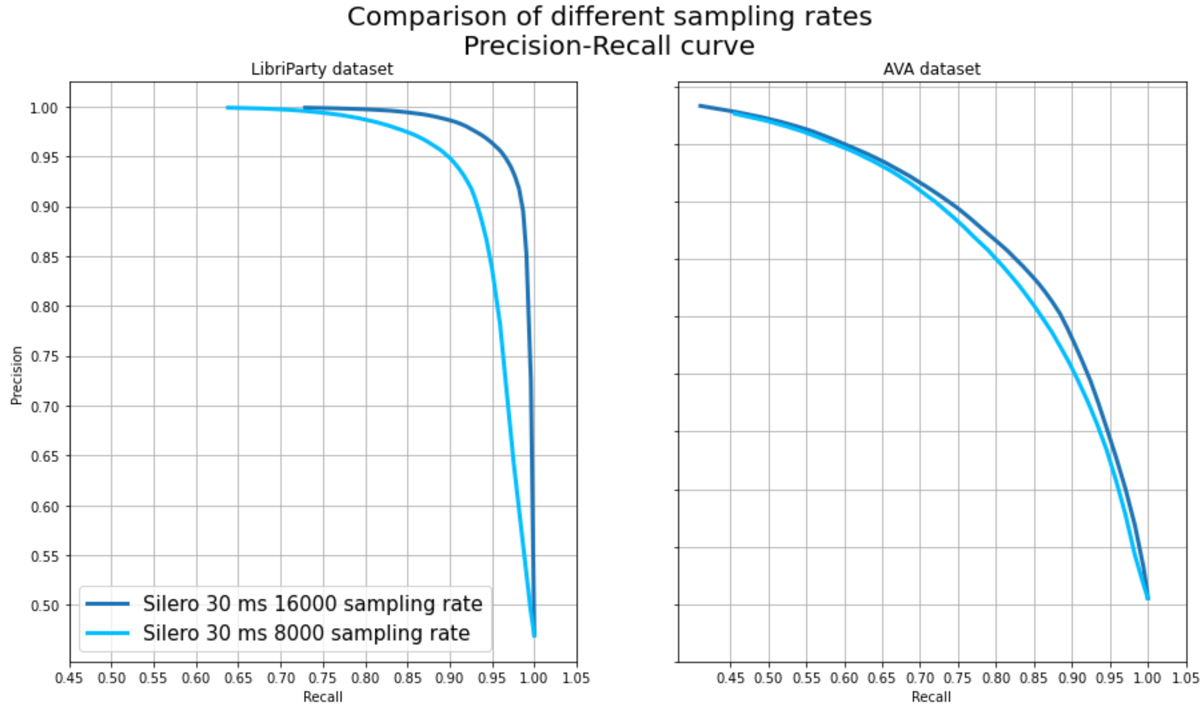
Silero VAD Vs Old Silero VAD

Parameters: `16000 Hz` sampling rate, `30 ms` (512 samples).

As you can see, there was a huge jump in the model's quality.



Sample Rate Comparison



For more information you can check the repository using the link below

[link to Silero VAD]

```
# BONUS 1: This cell contains the function for the gradio app
import os
import stable_whisper


def vid_to_subs(link):
    try:
        # Download video and extract audio


        # Construct the full path to the downloaded video
        video_path = download_video(link)[0]

        # Transcribe the audio
        model = stable_whisper.load_model('medium')
        result = model.transcribe(video_path)

        # Store transcription result in a variable

        transcription = result.to_txt()

        return transcription

    except Exception as e:
        print("An error occurred:", str(e))
```

Bonus 2

Hypothesis: By leveraging a ground-truth transcript, we can improve the accuracy and quality of the generated transcripts by using it as a reference or constraint during the transcription process or as a post-processing step. Approach:

Alignment: Align the ground-truth transcript with the audio file. This can be done using forced alignment techniques or by leveraging the timestamps from the generated transcript . The goal is to establish a mapping between the ground-truth text and the corresponding audio segments.

Transcript Segmentation: Segment the ground-truth transcript into smaller units, such as sentences or phrases, based on the alignment with the audio. This will allow for more granular comparisons and adjustments.

Confidence Scoring: For each segment of the generated transcript, calculate a confidence score based on various factors, such as acoustic model scores, language model scores, or word error rates compared to the ground-truth segment. Hybrid Transcript Generation:

For segments with high confidence scores (i.e., high accuracy compared to the ground-truth), retain the generated transcript segment as-is. For segments with low confidence scores, replace or adjust the generated transcript segment with the corresponding ground-truth segment.

Constrained Decoding: Alternatively, we could use the ground-truth transcript as a constraint during the decoding process of the speech recognition system. This approach would involve biasing the decoder to favor hypotheses that are closer to the ground-truth transcript, effectively incorporating the ground-truth information into the transcription process itself.

Iterative Refinement: Optionally, we could iterate the process of transcript generation and refinement, using the improved transcript from the previous step as the new ground-truth for the next iteration. This could potentially lead to further improvements in accuracy.

The main hypothesis behind this approach is that by leveraging the ground-truth transcript, which is assumed to be highly accurate, we can correct or adjust the generated transcript in regions where it deviates significantly from the ground-truth. This can help mitigate errors introduced by the speech recognition system, especially in challenging acoustic conditions or for less common words or phrases.

It's important to note that the effectiveness of this approach will depend on the quality and accuracy of the ground-truth transcript itself, as well as the ability to accurately align it with the audio. Additionally, care must be taken to ensure that the ground-truth transcript does not introduce its own errors or biases into the final transcript.

For the specific example of improving the transcription quality of segments using the transcript scraped from the provided link, the same approach can be applied. The scraped transcript can be treated as the ground-truth, and the steps outlined above can be followed to leverage it for improving the accuracy of the generated transcripts for those segments.

```
#@markdown # Gradio App
#@markdown Run this cell to start the gradio app the app take
import gradio as gr


demo = gr.Interface(fn=vid_to_subs, inputs="textbox", outputs

demo.launch(share=True, debug=True)
```

Gradio App

Run this cell to start the gradio app the app takes the youtube video link and generates the substitles for it.

```
Colab notebook detected. This cell will run indefinitely so that you can see errors and logs. To turn off, set debug=Fal
Running on public URL: https://71e8e60ffe75355e0c.gradio.live

This share link expires in 72 hours. For free permanent hosting and GPU upgrades, run `gradio deploy` from Terminal to d
```

gradio

No interface is running right now

```
Download complete!
MoviePy - Writing audio in /content/./video.mp3
MoviePy - Done.
Audio extracted successfully at: /content/./video.mp3
Transcribe:   0%|          | 0/690.89 [00:02<?, ?sec/s]Detected language: english
Transcribe: 100%|██████████| 690.89/690.89 [02:23<00:00,  4.81sec/s]
Keyboard interruption in main thread... closing server.
Killing tunnel 127.0.0.1:7860 <> https://71e8e60ffe75355e0c.gradio.live
```

Testing webrtc and other models

The code cells in this block is just me checking other models like stable whisper vs whisper transcription and webrtc for task 1 to see how they compare

```
import whisper
model = whisper.load_model("base")

result = model.transcribe("/content/Sarvam AI Wants To Leverage AI In Health & Education Says Co Founder Vivek Raghavan With

with open("transcription.txt", "w") as f:
  f.write(result["text"])
```

```
result
```

```
import stable_whisper
model = stable_whisper.load_model('medium')
result = model.transcribe('/content/Sarvam AI Wants To Leverage AI In Health & Education Says Co Founder Vivek Raghavan With
result.to_srt_vtt('audio_med.srt')
result.to_txt('audio_med.txt')
```

```
result.to_txt('audio_med.txt')
```

```
!pip install webrtcvad
!pip install SpeechRecognition
!pip install pydub
!pip install pysrt
!pip install spacy
!python -m spacy download en_core_web_sm
```

```
import webrtcvad
import speech_recognition as sr
import pysrt
import spacy
import datetime
import collections
import numpy as np
import wave
from typing import List, Tuple

# Load the spaCy model for sentence boundary detection
nlp = spacy.load("en_core_web_sm")

def parse_srt(srt_file: str) -> List[Tuple[str, float, float]]:
    """
    Parse the .srt file and return a list of (text, start_time, end_time) tuples.
    """
    subs = pysrt.open(srt_file)
    parsed_subs = []
    for sub in subs:
        start_datetime = datetime.datetime.combine(datetime.date.today(), sub.start.to_time())
        end_datetime = datetime.datetime.combine(datetime.date.today(), sub.end.to_time())
        start_time = start_datetime.timestamp()
        end_time = end_datetime.timestamp()
        parsed_subs.append((sub.text, start_time, end_time))
    return parsed_subs
def detect_speech_regions(audio_file: str) -> List[Tuple[float, float]]:
    """
    Detect voice activity regions in the audio file.
    """
    vad = webrtcvad.Vad()
    speech_regions = []

    with wave.open(audio_file, 'r') as wav:
        rate = wav.getframerate()
        frames = wav.getnframes()
        audio_data = np.frombuffer(wav.readframes(frames), dtype=np.int16)

        window_duration = 0.03  # 30ms
        window_size = int(rate * window_duration)
        windows = [audio_data[i:i + window_size] for i in range(0, len(audio_data), window_size)]

        speech_start = None
        for i, window in enumerate(windows):
            is_speech = vad.is_speech(bytes(window), rate)
            if is_speech:
                if speech_start is None:
                    speech_start = i * window_duration
            else:
                if speech_start is not None:
                    speech_end = (i + 1) * window_duration
                    speech_regions.append((speech_start, speech_end))
                    speech_start = None

    return speech_regions
def semantic_chunking(srt_file: str, audio_file: str, max_chunk_duration: float = 15.0) -> List[dict]:
    """
    Perform semantic chunking on the transcript and audio file.
    """
    parsed_subs = parse_srt(srt_file)
    speech_regions = detect_speech_regions(audio_file)

    chunks = []
    current_chunk = []
    current_chunk_start = None
    current_chunk_end = None

    for text, start_time, end_time in parsed_subs:
        doc = nlp(text)
        for sent in doc.sents:
            sent_start = start_time + sent.start_char / len(text)
            sent_end = start_time + sent.end_char / len(text)
            overlaps_speech = any(
                speech_start <= sent_start <= speech_end or speech_start <= sent_end <= speech_end
                for speech_start, speech_end in speech_regions
            )
            if overlaps_speech:
                if not current_chunk:
                    current_chunk_start = sent_start
                current_chunk.append(sent.text)
                current_chunk_end = sent_end
                if (current_chunk_end - current_chunk_start) >= max_chunk_duration:
```

```
                chunk_id = len(chunks) + 1
                chunk_text = " ".join(current_chunk)
                chunk_length = current_chunk_end - current_chunk_start
                chunks.append({
                    "chunk_id": chunk_id,
                    "text": chunk_text,
                    "chunk_length": chunk_length,
                    "start_time": current_chunk_start,
                    "end_time": current_chunk_end,
                })
                current_chunk = []
                current_chunk_start = None
                current_chunk_end = None

    if current_chunk:
        chunk_id = len(chunks) + 1
        chunk_text = " ".join(current_chunk)
        chunk_length = current_chunk_end - current_chunk_start
        chunks.append({
            "chunk_id": chunk_id,
            "text": chunk_text,
            "chunk_length": chunk_length,
            "start_time": current_chunk_start,
            "end_time": current_chunk_end,
        })

    return chunks

from pydub import AudioSegment

def convert_mp3_to_wav(mp3_file, wav_file):
    # Load the MP3 file
    audio = AudioSegment.from_mp3(mp3_file)
    # Export the audio to WAV
    audio.export(wav_file, format="wav")
```

```
!pip install -q torchaudio

SAMPLING_RATE = 16000

import torch
torch.set_num_threads(1)

from IPython.display import Audio
from pprint import pprint
```

```
convert_mp3_to_wav("/content/Sarvam AI Wants To Leverage AI In Health & Education Says Co Founder Vivek Raghavan With OpenHa
```

```
# Example usage
srt_file = "/content/audio_med.srt"
audio_file = "/content/Sarvam AI Wants To Leverage AI In Health & Education Says Co Founder Vivek Raghavan With OpenHathi.mp
chunked_output = semantic_chunking(srt_file, audio_file)
print(chunked_output)
```

```
chunks = semantic_chunking('audio_med.srt', 'output.wav')
print(chunks)
```

```
convert_mp3_to_wav("/content/Sarvam AI Wants To Leverage AI In Health & Education Says Co Founder Vivek Raghavan With OpenHa
```

```
!whisper "/content/Sarvam AI Wants To Leverage AI In Health & Education Says Co Founder Vivek Raghavan With OpenHathi.mp4" -
```

Show hidden output

```
!whisper "/content/Sarvam AI Wants To Leverage AI In Health & Education Says Co Founder Vivek Raghavan With OpenHathi.mp4" -
```

Show hidden output

Task 2: Exploratory Data Analysis of New Testament Audio and Text

Problem Statement:

The objective of this task is to conduct a comprehensive exploratory data analysis (EDA) on the audio and text data of the 260 chapters of the New Testament in your mother tongue (excluding English). The data should be obtained through web scraping from [Faith Comes By Hearing](#).

The workflow for this task should include:

1. **Web Scraping:** Systematically download the audio files and their corresponding textual content for each of the 260 chapters of the New Testament from the specified website.
2. **Data Preparation:** Organize the data by chapters, ensuring each audio file is matched with its corresponding text.
3. **Exploratory Data Analysis:** Analyze the data to uncover patterns.

Run this cell to install the necessary libraries

Show code

Show hidden output

Web Scraping

Method Used:

I gathered the links for all the chapters of the new testament and then scraped them using Beautiful Soup and Selenium.

Alternatively we could also use the API available in the sites code and ping it for all the data collected. The reason why I didn't do that was because the with the first method you could directly get the text and the audio files whereas if I were to use the api I would have to ping it get the response back and then would have to extrapolate from the json response. It's just one additional step which would add a layer of complexity to it hence I took the first approach

```
# Code to setup the sesssion
# Import necessary libraries
from bs4 import BeautifulSoup
import requests
import tls_client

from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import time

session = tls_client.Session(
    client_identifier='chrome119',
    random_tls_extension_order=True,
)

# Set up the Chrome driver (you may need to download the appropriate version for your system)
chrome_options = webdriver.ChromeOptions()
chrome_options.add_argument('--headless')  # Run Chrome in headless mode (without a GUI)
chrome_options.add_argument('--no-sandbox')  # Required for running in Colab
chrome_options.add_argument('--disable-dev-shm-usage')  # Required for running in Colab
```

Data Preparation

The data is scraped in such a way that the text is stored along with it's audio maintaing the data integrity.

```
# Scraping function

import requests
from bs4 import BeautifulSoup
import pandas as pd


# Function to scrape data for a chapter
def scrape_chapter(url):
    response = requests.get(url)

    if response.status_code == 200:

        driver = webdriver.Chrome(options=chrome_options)

        # Navigate to the website
        driver.get(url)

        # Wait for the page to load
        time.sleep(2)

        # Extract the desired data from the website
        html_content = driver.page_source

        # Print the extracted data
        print(html_content)

        # Close the browser
        driver.quit()
        soup = BeautifulSoup(response.text, 'html.parser')
        soup2 = BeautifulSoup(html_content, 'html.parser')

        spans_with_verseid = soup.find_all('span', attrs={'data-verseid': True})

        text = ""
        for span in spans_with_verseid:
            line = span.text.strip()
            text += line + "\n"

        audio_url = soup2.find('video', class_='audio-player')['src']

        return text.strip(), audio_url

    else:
        print('Failed to retrieve the webpage. Status code:', response.status_code)
        return None, None
```

Chapter URLS

Run this cell to intialize all the chapter urls

Show code

```
# Combining all the urls into a single list

# df.to_csv('matthew.csv')
all_urls = [
    matthew_chapter_urls,
    mark_chapter_urls,
    luke_chapter_urls,
    john_chapter_urls,
    acts_chapter_urls,
    romans_chapter_urls,
    corinthians1_chapter_urls,
    corinthians2_chapter_urls,
    galatians_chapter_urls,
    ephesians_chapter_urls,
    philippians_chapter_urls,
    colossians_chapter_urls,
    thessalonians1_chapter_urls,
    thessalonians2_chapter_urls,
    timothy1_chapter_urls,
    timothy2_chapter_urls,
    titus_chapter_urls,
    phm_chapter_urls,
    hebrews_chapter_urls,
    james_chapter_urls,
    peter1_chapter_urls,
    peter2_chapter_urls,
```

```
        peter2_chapter_urls,
        john1_chapter_urls,
        john2_chapter_urls,
        john3_chapter_urls,
        judah_chapter_urls,
        revela_chapter_urls
    ]
```

```
# Implemented threads to get the scraping done in an efficient manner

import threading
import time
import pandas as pd
import requests
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.util.retry import Retry
import random
import socks
import socket

# Function to scrape data for a single chapter
def scrape_chapter_data(url):
    text, audio_url = scrape_chapter(url)
    if text is not None and audio_url is not None:
        return (text, audio_url)
    else:
        return None

# Number of threads to use
num_threads = 5
# Delay between each request (in seconds)
request_delay = 5
# Maximum number of retry attempts
max_retry_attempts = 3

# Combine all URL lists into a single list
all_urls = (matthew_chapter_urls + mark_chapter_urls + luke_chapter_urls + john_chapter_urls +
            acts_chapter_urls + romans_chapter_urls + corinthians1_chapter_urls + corinthians2_chapter_urls +
            galatians_chapter_urls + ephesians_chapter_urls + philippians_chapter_urls + colossians_chapter_urls +
            thessalonians1_chapter_urls + thessalonians2_chapter_urls + timothy1_chapter_urls +
            timothy2_chapter_urls + titus_chapter_urls + phm_chapter_urls + hebrews_chapter_urls +
            james_chapter_urls + peter1_chapter_urls + peter2_chapter_urls + john1_chapter_urls +
            john2_chapter_urls + john3_chapter_urls + judah_chapter_urls + revela_chapter_urls)

# Split the list of all URLs into chunks
url_chunks = [all_urls[i:i + len(all_urls) // num_threads] for i in range(0, len(all_urls), len(all_urls) // num_threads)]

# List to store tuples of text and audio URL for each chapter
chapter_data_list = []

# Function to be executed by each thread
def worker(chunk):
    for url in chunk:
        data = scrape_with_retry(url)
        if data:
            chapter_data_list.append(data)
        time.sleep(request_delay)  # Add delay between requests

# Define a retry decorator
def retry(max_attempts=max_retry_attempts):
    def decorator(func):
        def wrapper(*args, **kwargs):
            attempts = 0
            while attempts < max_attempts:
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    attempts += 1
                    print(f"Attempt {attempts}/{max_attempts} failed for URL: {args[0]}. Error: {e}")
                    time.sleep(2)  # Wait before retrying
            print(f"Maximum retry attempts reached for URL: {args[0]}")
            return None
        return wrapper
    return decorator

# Apply retry decorator to scrape_chapter_data function
scrape_with_retry = retry()(scrape_chapter_data)

# Create and start threads
threads = []
for chunk in url_chunks:
    thread = threading.Thread(target=worker, args=(chunk,))
    thread.start()
    threads.append(thread)

# Wait for all threads to complete
for thread in threads:
    thread.join()

# Create a pandas DataFrame from the list of chapter data
```

```
df = pd.DataFrame(chapter_data_list, columns=['Text', 'Audio_URL'])

# Display the DataFrame
print(df)
```

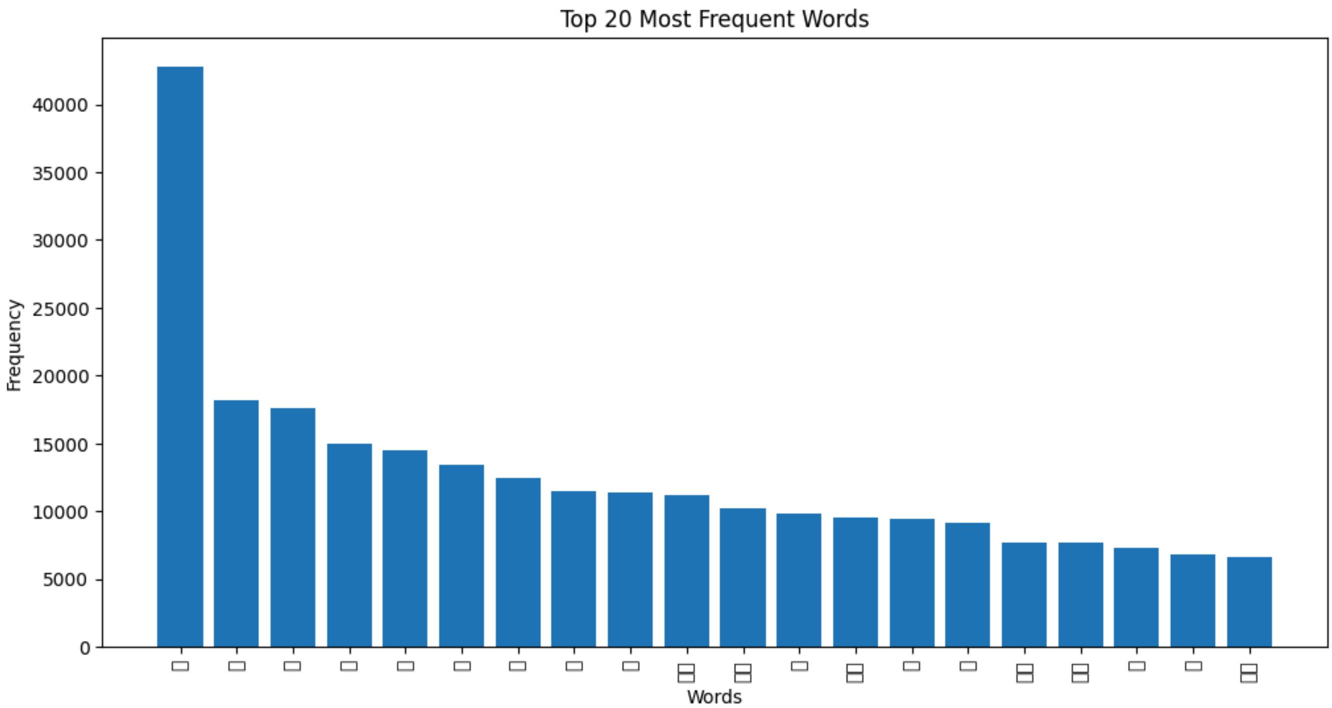
Show hidden output

```
df
```

Show hidden output

```
import pandas as pd
import matplotlib.pyplot as plt
import regex as re
from collections import Counter

# Load the data into a pandas DataFrame
df = pd.read_csv('alll.csv')

# Convert 'Text' column to string
df['Text'] = df['Text'].astype(str)

# Concatenate all text from the first column
text = ' '.join(df['Text'])

# Remove non-Devanagari characters
devanagari_pattern = re.compile(r'[^\u0900-\u097F]+')
text = devanagari_pattern.sub('', text)

# Tokenize the text into words
words = re.findall(r'\X', text)

# Count word frequencies
word_freq = Counter(words)

# Sort the words by frequency in descending order
sorted_words = sorted(word_freq.items(), key=lambda x: x[1], reverse=True)

# Get the top 20 most frequent words
top_words = sorted_words[:20]

# Create a bar plot
plt.figure(figsize=(12, 6))
plt.bar(range(len(top_words)), [freq for word, freq in top_words])
plt.xticks(range(len(top_words)), [word for word, freq in top_words], rotation=90)
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.title('Top 20 Most Frequent Words')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2352 (\N{DEVANAGARI LETTER RA
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Matplotlib currently does not suppo
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2360 (\N{DEVANAGARI LETTER SA
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2325 (\N{DEVANAGARI LETTER KA
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2344 (\N{DEVANAGARI LETTER NA
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2346 (\N{DEVANAGARI LETTER PA
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2361 (\N{DEVANAGARI LETTER HA
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2313 (\N{DEVANAGARI LETTER U}
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2324 (\N{DEVANAGARI LETTER AU
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2350 (\N{DEVANAGARI LETTER MA
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2375 (\N{DEVANAGARI VOWEL SIG
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2404 (\N{DEVANAGARI DANDA}) m
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2357 (\N{DEVANAGARI LETTER VA
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2351 (\N{DEVANAGARI LETTER YA
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2367 (\N{DEVANAGARI VOWEL SIG
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2366 (\N{DEVANAGARI VOWEL SIG
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2348 (\N{DEVANAGARI LETTER BA
  fig.canvas.print_figure(bytes_io, **kw)
/usr/local/lib/python3.10/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 2340 (\N{DEVANAGARI LETTER TA
  fig.canvas.print_figure(bytes_io, **kw)
```




```
print(top_words)
```

```
[('र', 42786), ('स', 18188), ('क', 17564), ('न', 14948), ('प', 14516), ('ह', 13452), ('उ', 12448), ('औ', 11504), ('म', 1
```

```
import pandas as pd
import re
import string

# Load the custom Hindi stopwords from a file
with open('stop_hindi.txt', 'r', encoding='utf-8') as f:
    hindi_stopwords = f.read().splitlines()

# Function to preprocess text
# Function to preprocess text
# Function to preprocess text
def preprocess_text(text):
    # Convert input to string if necessary
    text = str(text)

    # Remove unwanted characters
    text = re.sub(r'[^ा-ॐ\s]', '', text)

    # Tokenize the text
    tokens = text.split()

    # Remove stopwords
    filtered_tokens = [word for word in tokens if word not in hindi_stopwords]

    # Join the tokens back into a string
    preprocessed_text = ' '.join(filtered_tokens)

    return preprocessed_text

# Apply text preprocessing and calculate text statistics
df['Preprocessed_Text'] = df['Text'].apply(preprocess_text)
df['Word_Count'] = df['Preprocessed_Text'].apply(lambda x: len(x.split()))
df['Unique_Word_Count'] = df['Preprocessed_Text'].apply(lambda x: len(set(x.split())))
df['Avg_Word_Length'] = df['Preprocessed_Text'].apply(lambda x: sum(len(word) for word in x.split()) / max(len(x.split()), 1

# Analyze the distributions of text metrics
print("Word Count Distribution:")
print(df['Word_Count'].describe())

print("\nUnique Word Count Distribution:")
print(df['Unique_Word_Count'].describe())

print("\nAverage Word Length Distribution:")
print(df['Avg_Word_Length'].describe())
```

```
Word Count Distribution:
count     183.000000
mean     1144.557377
std       569.705395
min         0.000000
25%       825.000000
50%      1128.000000
75%      1470.000000
max      2640.000000
Name: Word_Count, dtype: float64

Unique Word Count Distribution:
count    183.000000
mean      69.497268
std       24.584877
min        0.000000
25%       62.000000
50%       73.000000
75%       84.500000
max      118.000000
Name: Unique_Word_Count, dtype: float64

Average Word Length Distribution:
count    183.000000
mean       1.371865
std        0.400850
min        0.000000
25%        1.428103
50%        1.474836
75%        1.519201
max        1.694853
Name: Avg_Word_Length, dtype: float64
```

```
# Warning this cell took about an hour to run so try to avoid it
import pandas as pd
from pydub import AudioSegment
import librosa
import requests

# Function to download and extract audio features
def extract_audio_features(audio_url):
    # Download the audio file
    if audio_url.startswith('_'):
        return None
    response = requests.get(audio_url)

    # Check if the download was successful
    if response.status_code == 200:
        # Save the downloaded audio data to a temporary file
        with open("temp.mp3", "wb") as f:
            f.write(response.content)

        # Load the audio data using librosa
        audio_data, sample_rate = librosa.load("temp.mp3")

        # Calculate audio duration
        audio_length = len(audio_data) / sample_rate

        # Calculate audio bit rate
        audio_segment = AudioSegment.from_file("temp.mp3", format="mp3")
        bit_rate = audio_segment.frame_rate * audio_segment.sample_width * 8

        # Extract other audio features
        # For example, you can extract Mel-Frequency Cepstral Coefficients (MFCCs)
        mfccs = librosa.feature.mfcc(y=audio_data, sr=sample_rate)

        return audio_length, sample_rate, bit_rate, mfccs
    else:
        return None

# Apply audio feature extraction
df['Duration'] = df['Audio_URL'].apply(lambda url: extract_audio_features(url)[0] if extract_audio_features(url) is not None
df['Sample_Rate'] = df['Audio_URL'].apply(lambda url: extract_audio_features(url)[1] if extract_audio_features(url) is not N
df['Bit_Rate'] = df['Audio_URL'].apply(lambda url: extract_audio_features(url)[2] if extract_audio_features(url) is not None
df['MFCCs'] = df['Audio_URL'].apply(lambda url: extract_audio_features(url)[3] if extract_audio_features(url) is not None el

# # Analyze the distributions of audio features
print("Audio Duration Distribution:")
print(df['Duration'].describe())

print("\nSample Rate Distribution:")
print(df['Sample_Rate'].describe())

print("\nBit Rate Distribution:")
print(df['Bit_Rate'].describe())
```

```
Audio Duration Distribution:
count    181.000000
mean     305.872990
std      111.686024
min      116.472018
25%      227.424036
50%      284.736009
75%      369.744036
max      636.336009
Name: Duration, dtype: float64

Sample Rate Distribution:
count      181.0
mean     22050.0
std          0.0
min      22050.0
25%      22050.0
50%      22050.0
75%      22050.0
max      22050.0
Name: Sample_Rate, dtype: float64

Bit Rate Distribution:
count       181.0
mean     384000.0
std           0.0
min      384000.0
25%      384000.0
50%      384000.0
75%      384000.0
max      384000.0
```