Big Data Analytics 2024 Project 1

Technical Report

By Diego Minaya

Summary

The project succesfully completes all the processing tasks for videos including downloading, extracting audio, transcription, sentiment analysis, translation, and emotion detection. Each stage is supported by a dedicated Python class, leveraging libraries like moviepy, speech_recognition, TextBlob, googletrans, spaCy, nltk, and NRCLex.

To enhance efficiency and manage concurrent tasks, the project employs multithreading, a semaphore to limit concurrent downloads, and a mutex to ensure the safe access to shared resources.

Table of Contents

Download_URL

Description

First i used a sequential script to download each video from the urls I saved in video_urls.txt. The following code saves all the urls in a list:

```
url_list = []
with open("Video_Analysis/video_urls.txt", "r") as f:
    for line in f:
        url_list.append(line)
```

To use parallel programming, I decided to use threads instead of processes since downloading 15 videos do not require much processing power and they can be easily controlled with semaphores ,to limit the number of downloads that happen at the same time, and mutex, to restric access to download_log.txt and provent multiple threads from trying to write to it at the same time. Below you can see my implementation using a mutex and a semaphore:

```
limiter = threading.BoundedSemaphore(5)
data_lock = threading.Lock() # mutex

def download_video(url, index):
    limiter.acquire()
    yt = YouTube(url)
    stream = yt.streams.get_highest_resolution()
    print(f"Thread {index}: Downloading video: {yt.title}")
    stream.download(output_path="video_output")
    print(f"Thread {index} : Download completed: {yt.title}")
    current_timestamp = datetime.datetime.now()
    current_timestamp = current_timestamp.strftime('%H:%M, %d %B %Y')
    thread_log = Logger(yt.title,index,current_timestamp,url,True)
    print(thread_log.getData())
    data_lock.acquire()
    thread_log.recordData()
    data_lock.release()
    limiter.release()
    Return
```

This function also saves some extra information for another class called Logger, which I will explain later. Finally this download_video function is called using a list where all the threads are saved and the threads are started later in another for loop:

```
## Parallel Download videos from the list
downloadThreads = []
for i in range(len(url_list)):
    thread = threading.Thread(target=download_video, args=(url_list[i],i,))
    downloadThreads.append(thread)
    thread.start()

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

Time and Space Complexity

To measure the performance of both the sequential and parallel code I used the timeit library. The sequential code took 11.55 seconds to download all the 15 videos. The parallel code took 19.53848361968994 seconds seconds to download all the 15 videos.

The time complexity of both algorithms is linear, O(n), since they have to go trough all the urls that are saved in a list.

However the space complexity of the parallel code is O(n) since it has to create a new thread for each video that is downloaded while the space complexity of the sequential code is O(1) since it only uses variables and no lists.

Download_Log

I created a different class called Logger. It captures information about each downloaded video including: the video name, the thread that downloaded it, the timestamp of the download, the URL of the video, and whether the download was successful. It also saves all this information in a text file called download_log.txt

As I explained in thre previous section I used a mutex that restricts the access of threads to the text file. The code of the logger class is very simple:

```
class Logger:
    def __init__(self,videoName, thread, timeStamp, URL, Downloaded):
        self.videoName = videoName
        self.thread = thread
        self.timeStamp = timeStamp
        self.URL = URL
        self.Downloaded = Downloaded

    def getData(self):
        return self.videoName,self.thread,self.timeStamp,self.URL,self.Downloaded

    def recordData(self):
        with open('download_log.txt', 'a') as file:
            file.write(f"Timestamp: {self.timeStamp} Name of video:
{self.videoName} Download by thread #: {self.thread} URL: {self.URL} Download:
{self.Downloaded}\n")
```

Video_Analysis

Before all the scripts below I used a small for loop to get the path for each of the videos downloaded:

```
#check the videofiles
videofiles = [f for f in listdir('video_output') if isfile(join('video_output',
f))]
```

I did this because I was getting some errorrs regarding the video paths.

Extract_Audio

For most of the scripts in this section I created a paralell programming and sequential programming version. For example here is the sequential version to extract audio:

```
#Serially extract audio from the downloaded videos
for video in videofiles:
    audioExtractor = AudioExtractor()
    video_path = join("video_output", video)  # Ensure full path is passed
    try:
        audioExtracted.append(audioExtractor.extractAudio(video_path))
    except Exception as e:
        print(f"Error extracting audio from {video_path}: {e}")
```

In this version the AudioExtractor class is created for every videopath and once the function extractAudio finishes, it saves the path of the audio extracted inside a list called audioExtracted.

The asynchronous version is very similar to the sequential one, but instead of creating a new AudioExtractor for every video, I created a list of threads of AudioExtractor classes and used them to extract audio from the videos asynchronously.

```
#Parallel extract audio from the downloaded videos
for i in range(len(videofiles)):
    video_path = join("video_output", videofiles[i])  # Ensure full path is passed
    thread = threading.Thread(target=AudioExtractor().extractAudio, args=
(video_path,audioExtracted,))
    audioExtractThreads.append(thread)
    thread.start()

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

Here I realized that using thread.join is very important because even though I can initialize threads without it, it makes sure that the threads complete their tasks before running the next line of code. I also would like to mention that extracting the audio from video did not seem to be heavyweight for my machine so I decided to keep using threads instead of processses.

My class AudioExtractor is very simple:

```
import moviepy.editor as mp
import os

class AudioExtractor:
    def __init__(self):
        self.output_folder = "extracted_audio"
        # Create the folder if it does not exist
        if not os.path.exists(self.output_folder):
            os.makedirs(self.output_folder)

    #replace the mp4 extension with wav
    def replaceMP4(self,str):
        return str[:-4] + '.wav' if str.endswith('.mp4') else str
```

```
    #Extract the audio from the video
    def extractAudio(self, videoPath,audioExtracted):
        video = mp.VideoFileClip(videoPath)
        name = video.filename
        name = os.path.basename(video.filename)  # Get the base name of the file
        audio_path = os.path.join(self.output_folder, self.replaceMP4(name))  #
Create the path in the "extracted_audio" folder
        video.audio.write_audiofile(audio_path)
        audioExtracted.append(audio_path)
        return audio_path
```

In the constructor of the class I defined the folder where all the extracted audio will be saved, if it does not exists, it will create it.

Then, the method replaceMP4 replaces the extension of the name of the file with wav.

Finally the method to extract the Audio uses the path of the video file to get the audio and it extracts it with the moviepy.editor library and the new path is saved in a list. This list will be used for other tasks.

The other classes are very similar to this one.

Transcribe_Audio

The class AudioTranscriber writes the text based on an audio file and saves the result in a folder. I got a lot of problems using the library needed for this since Google Web Speech API told me that there were some bad requests for some files but I fixed it by limiting the duration of the audio files to only 3 minutes. I used threats for this class as well since it relies in remote processing and not local processing power.

The code is very similar to the previous class but this time instead of saving it as a .wav file it changes it to txt and uses the speech_recognition library to extract the text from the audio.

Sentiment_Analysis

SentimentAnalyser performs sentiment analysis on text files. It reads text content from a file, analyzes the sentiment using the TextBlob library, and saves the sentiment results to a new file.

Translate_Text

In the Tranlator file there is a SpanishTranslator class that uses the Google Translate API through the googletrans library. I tried to use TextBlob but it didnt work.

Extract_Emotions

Finally my Emotion extractor class analyzes the emotions in a text file and save the results to a new file. This class uses libraries like spaCy for natural language processing, nltk for tokenization, and NRCLex for emotion detection. At the beginning of running the code it downloads the punkt package.