# Big Data Analytics Coursework

# CineSense Video Analysis

Contents

[Big Data Analytics Coursework 1](#_Toc170140263)

[CineSense Video Analysis 1](#_Toc170140264)

[Introduction 3](#_Toc170140265)

[Downloading the videos and creating logs: 3](#_Toc170140266)

[Space and time complexity of download\_videos\_and\_log function: 5](#_Toc170140267)

[Audio extraction 5](#_Toc170140268)

[Text Extraction: 7](#_Toc170140269)

[Extract Sentiments: 7](#_Toc170140270)

[Translate Text: 8](#_Toc170140271)

[Extract Emotions: 9](#_Toc170140272)

[Folder Structure: 9](#_Toc170140273)

# Introduction

The objective of this project is to develop a python script to download and analyze YouTube videos. This project compares the use of multiprocessing for parallel execution of these tasks with serial execution.

The project can be divided into following tasks: Downloading the videos and creating log, Extraction of audio, extracting text, extracting sentiments, translating the text and extracting emotions.

# Downloading the videos and creating logs:

The **download\_videos.py** file contains the code to download the videos.

The **read\_urls** function reads the file with the URLs of the videos to download and returns a list of the URLs.

def read\_urls(file\_name):

    urls=[]

    with open(file\_name,'r') as file:

        for line in file:

            urls.append(line)

    return urls

The **download\_videos\_and\_log** function downloads a single video and creates a log in the log\_file.txt. This function has two parameters, thread ID and the URL of the video.

For this task, I have used threads as this is an I/O bound task and threads are faster than multiprocessing when it comes to I/O tasks. To make sure that only five videos are downloaded at the same time and only one thread writes to the logfile at a time semaphore and mutex locks are used, respectively.

The semaphore lock with five maximum threads is acquired before the video is downloaded making sure only five videos are downloaded at a time.

#download the video in the desired folder

    try:

        stream.download(output\_path=f'Output/{yt.title}' ,filename=f'{yt.title}.mp4')

        print(f' Thread {threadID} successfully downloaded video:{yt.title}')

If the download fails in the try block,  
the except block will print the error to the screen and write to the logfile with ‘download: False’ after acquiring a mutex lock, making sure only one thread writes to the logfile at a time.

#if any error,write into the logfile with 'download:False' and print the error on the screen

    except Exception as e:

        #aquring mutex lock

        mutex\_lock.acquire()

        #open the logfile to write the log

        with open('log\_file.txt','a') as logfile:

            today=datetime.datetime.now()

            date\_time= today.strftime("%H:%M %d-%m-%Y")

            logfile.write(f'Timestamp:{date\_time},URL:{url},Thread\_ID: Thread {threadID},download:{False}\n')

        #print the exception to the screen

            print(e)

If the video download is successful, the else block will write to the logfile with ‘download: True’ after mutex lock is acquired.

else:

        #aquiring mutex lock

        mutex\_lock.acquire()

        #open the logfile to write the log

        with open('log\_file.txt','a') as logfile:

            today=datetime.datetime.now()

            date\_time= today.strftime("%H:%M %d-%m-%Y")

            logfile.write(f'Timestamp:{date\_time},URL:{url},Thread\_ID: Thread {threadID},download:{True}\n')

The semaphore and mutex locks are released after writing to the logfile using the finally block.

#finally,release mutex and semaphore locks

    finally:

        mutex\_lock.release()

        semaphore\_lock.release()

The serial downloader function downloads all the videos from the URLs in the list returned by the read URLs function in serial. It prints the total time taken to download all the videos. Thread ID is set to one when the **download\_videos\_and\_log** function is called.

The **parallel\_download.py** file contains the function **parallel\_ThreadPoolDownloader** which is used to download the videos in parallel using threads. The semaphore lock ensures that only 5 videos are downloaded at a time and the mutex lock makes sure that only one thread writes to the logfile at a time.  
**concurrent.futures.ThreadPoolExecutor()** is used to download the videos in parallel. The function also prints the total time taken to download all the videos.

## Space and time complexity of download\_videos\_and\_log function:

Since the downloading of the video is the most significant operation in terms of space, the space complexity of the download\_videos\_and\_log function is O(N) with N being the length of the video to be downloaded. As the video length increases the space taken increases linearly with it.

The time complexity of the function is O(N) as well. Since, downloading the video is the most significant operation in terms of time as well. N is the length of the video. As the length of the video increases the time taken to download the file increases linearly.

# Audio extraction

The file **extract\_audio.py** consists of the script to extract audio from the downloaded video.

The **get\_file\_path** function returns a list with all the paths of the video files which require the audio to be extracted.

#function to store the video path of all videos in a list

def get\_file\_path(folder\_path):

    file\_path\_list=os.listdir(folder\_path)

    for i,j in enumerate(file\_path\_list):

        file\_path\_list[i]=f'Output/{j}/{j}.mp4'

    return file\_path\_list

The **extract\_audio** function takes the path of one video and extracts audio for that one video and saves it in a designated folder for that video.

#function to extract the audio from the given video file path

def extract\_audio(file\_path):

    try:

        video=mp.VideoFileClip(file\_path)

        #save the audio file in the designated video folder

        video.audio.write\_audiofile(f'{file\_path[0:-4]}.wav')

    except Exception as e:

        print(e)

The **serial\_audio\_extraction** function extracts the audio from all the videos in a seral manner. It prints the total time taken to extract the audio from all the videos. It takes the list of the paths of all the videos as parameter.

#function to extract audio for all the video files in serial

def serial\_audio\_extraction(file\_path\_list):

    t1=time.perf\_counter()

    for i in file\_path\_list:

        extract\_audio(i)

    t2=time.perf\_counter()

    print('Time taken to extract audio in serial(secs):',round(t2-t1,2))

The file **parallel\_audio\_extraction.py** contains the script to extract audio in parallel. It compares the use of threads, multiprocessing and **concurrent.futures** for audio extraction.  
**parallel\_thread\_pool\_runner** function uses **ThreadPoolExecutor** to extract the audio. **parallel\_process\_pool\_runner** function uses ProcessPoolExector to extract the audio.  
**parallel\_multiprocesserrunner** function uses **multiprocessing.Pool** to extract the audio.  
**Parallel\_threadrunner** uses threads to extract the audio.

Since, extraction of audio is a processing task, multiprocessing performs better than threading. Therefore, **parallel\_process\_pool\_runner and parallel\_multiprocesserrunner** function performs better than the rest.

# Text Extraction:

**The get\_audio\_file\_path** function returns two lists, one with all the audio files, the other with the path of all the audio files.

The **chunk\_audio\_and\_extract\_text** function takes in an audio file path as a parameter and divides the audio file into smaller files of maximum length of one minute and then extracts the text from these files to write it to a text file which is then saved in the designated folder for the audio.

The audio files are divided into smaller files because the **recognize\_google** method from speech recognition library cannot extract text for files bigger than 10 mb.

This function divides the audio file, saves it in the temporary memory which is then used to extract the text. Using BytesIO we can save data in memory as if it were a file, not requiring to actually write data to the disk. In this way, extra space is not required save the smaller files.

The **serial\_text\_recognition** function extracts the text from all the audio files in serial.

The parallel\_extract\_text.py file contains the function **parallel\_extract\_text\_using\_threads** which extracts text from all the audio files using threads. Audio extraction is a CPU bound task although it appears that threads perform better than multiprocessing. Therefore, reading the audio file and writing the text to the text file is more of a bottleneck than the processing and hence threads are used.

# Extract Sentiments:

The **extract\_sentiment.py** file contains the script which is used to extract sentiments from the text file of the video.  
The function **extract­\_sentiment** takes one text file and extracts the sentiment and writes it to a different text file and saves it in the designated video folder.

The **text\_file\_path.py** file contains the function **get\_text\_file\_path** which returns two lists, one with all text files and other with the path of the text files.

The **serial\_sentiment** function extracts sentiments for all the files in serial. This function takes in the list of the text file names and the list of their paths returned by the **get\_text\_file\_path** function as parameters.

The parallel\_extact\_sentiment.py file contains the function **parallel\_sentiment\_threads** which is used to extract the sentiments from the file in parallel using threads.  
This function also takes in the list of the text file names and the list of their paths as parameters.

Since extracting sentiments is a CPU bound task, I assumed that multiprocessing would be faster than threading.  
However, it seems that reading and writing from the text files is a bigger bottleneck than the computing required to extract sentiment, hence threads perform better than multiprocessing.

# Translate Text:

The file **translate.py** contains the code to translate the text from English to Spanish.

The function **translate\_text** reads a file and writes the translated text to a different file in the same folder.

The **text\_file\_path.py** file contains the function **get\_text\_file\_path** which returns two lists, one with all text files and other with the path of the text files.

The **serial\_translate** function translates the text for all the files in serial. The **serial\_translate** function takes in the list of the text file names and the list of their paths returned by the **get\_text\_file\_path** function as parameters.

The parallel\_translate.py file contains the **parallel\_translate\_threads** function which takes in the list of the text file names and the list of their paths returned by the **get\_text\_file\_path** as parameters and translates text for all the files in parallel using threads.

Again, translating is a CPU bound task, but threads perform better as reading and writing to the file is a bigger bottleneck than translating the text. Hence, the use of threads.

# Extract Emotions:

The file **extract\_emotions.py** contains the code to translate the text from English to Spanish.

The function **extract\_emotions** reads a file and writes the extracted emotional analysis to a different file in the same folder.

The **text\_file\_path.py** file contains the function **get\_text\_file\_path** which returns two lists, one with all text files and other with the path of the text files.

The **serial\_emotions** function extracts emotions for all the files in serial. This function takes in the list of the text file names and the list of their paths returned by the **get\_text\_file\_path** function as parameters and translates text for all the files in serial.

The **parallel\_emotions.py** file contains the **parallel\_emotions\_threads** function which takes in the list of the text file names and the list of their paths returned by the **get\_text\_file\_path** as parameters and extracts emotions for all the files in parallel using threads.

Again, extracting emotions is a CPU bound task, but threads perform better as reading and writing to the file is a bigger bottleneck than translating the text. Hence, the use of threads.

# Folder Structure:

Each video is downloaded in the output folder with the video name as the folder name.

A screenshot of a computer

Description automatically generated

All the files of a video are generated in the designated folder of the video.

A screenshot of a computer

Description automatically generated