TECHNICAL REPORT

Introduction

This project implements a video file processing process that includes loading videos, extracting audio, converting audio to text, extracting emotions from text, sentiment analysis, and translating text into Spanish. During the tasks of loading video files and extracting audio from videos, it was proposed to test various programming strategies, including sequential execution, multi-threading, multi-processing and asynchronous programming, with justification for optimal execution as assessed by the developer.

1. Aims

- Develop a Python application using multiprocessing, threading or asynchronous programming concepts to download and analyze YouTube videos.
- Compare and contrast different solutions for serial and parallel processing executions

2. Objectives

- Implement video download, audio extraction, speech–to-text conversion, emotion extraction, sentiment analysis, and text translation modules using both functional and object-oriented approaches.
- Evaluate and optimize the application using different processing strategies.
- document the application and prepare report.

Input parameters:

- YouTube video 2-3 minutes long
- Video_urls.txt contents 10-15 urls.
- Within the 'video_output' directory, each video organized into its out folder.

Output:

A dedicated folder designated for each video.

Example_Video (folder name would be the video name):

- video name.mp4
- video name.wav
- video name.txt
- video name_sensitivity.txt
- video name_spanish.txt
- video name_emotions.txt

3. Technical implementation

Development tools include Visual Studio Code for the features it provides. The project was implemented using the Python programming language.

Libraries and tools used:

- Pytube, for downloading videos from YouTube.
- Moviepy, for audio processing and extraction.
- SpeechRecognition, for converting audio to text.
- Nrclex, for extracting emotions from text.
- SpaCy, for natural language analysis and processing (NLP).
- Nltk, for natural language and sentiment processing.
- Textblob, for sentiment analysis and text translation.
- Deep_translator, for translating text into various languages.
- Standard Python libraries for working with files: os, time, datetime, multiprocessing, concurrent.features, threading, re, asyncio, pathlib.

4. Project architecture

For effective modeling, analysis, and implementation of the project, we used a modular approach, organizing the project into distinct modules for each functionality.

Description of modules

- main.py: command line tool the entire video processing pipeline.
- load_data.py: download list of URLs from text file.
- download_video.py: download the video from the given URL using Pytube.
- audio_extractor.py: Extract audio from a video file and save it as a WAV file using Moviepy.
- text_extractor.py: Transcribe extracted audio to text using SpeechRecognition.
- emotion_extractor.py: Extracts emotions from the text using the Nrclex and SpaCy.
- sentiment_analysis.py: Perform the sentiment analysis using Ntlk and Textblob.
- translator.py: Translate the text into another language using Deep_translator.

To optimize project documentation and provide a more systematic approach to development, the project module interaction diagram (Scheme of interaction between project modules) shown in Fig. 1.

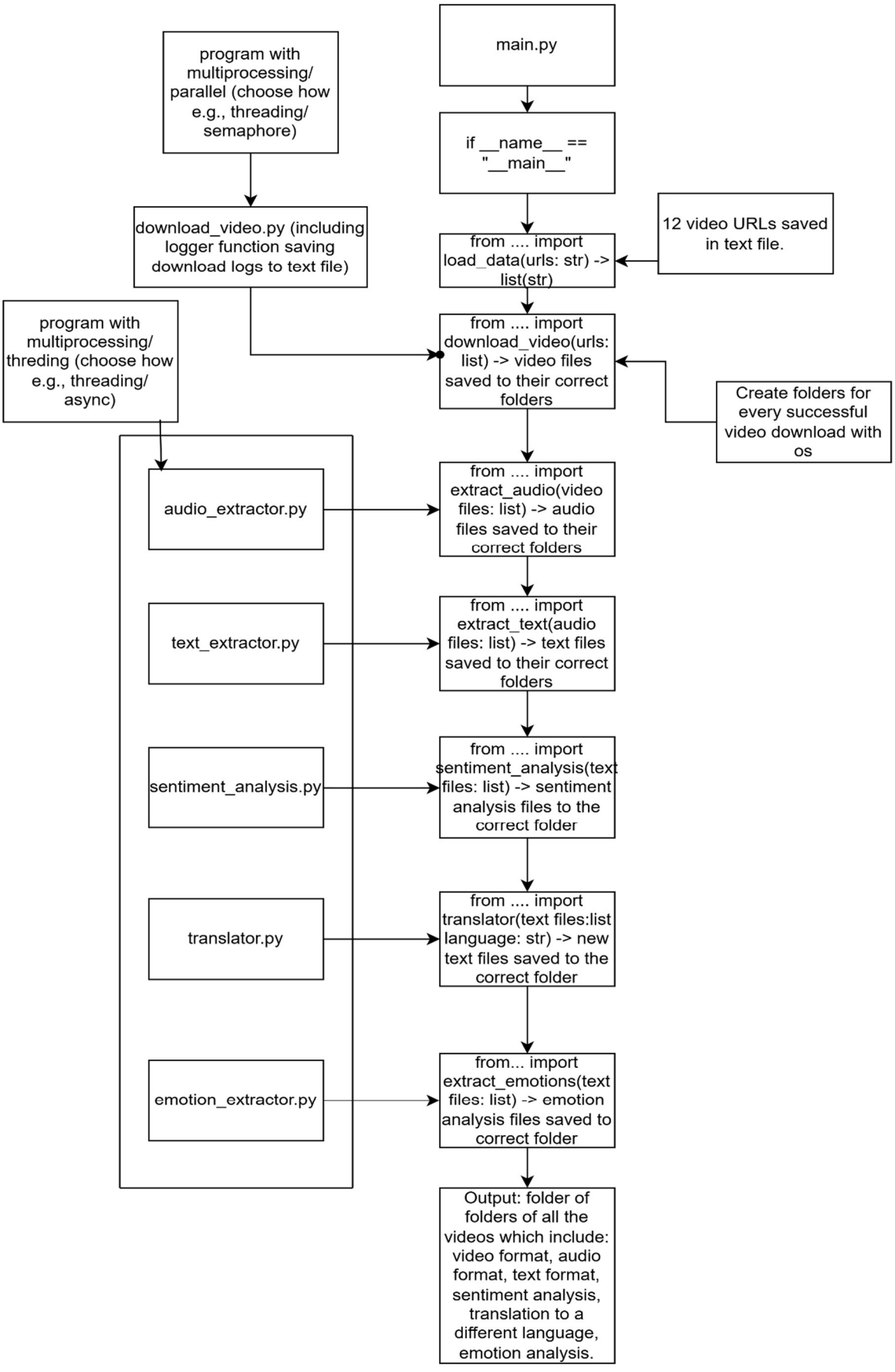

Fig1. Scheme of interaction between project modules

5. Programming strategy

- load_data.py

Strategy. The standard approach involves reading lines (URLs) from a file sequentially. This method is easy to code and effective when the number of URLs is limited.

Time complexity is O(n), where n- number of URLs.

Space complexity is O(n), all URLs are stored in a list.

- download_video.py

Strategy: The script provides multiple methods, including serial processing, multiprocessing and multithreading.

Time complexity:

Serial: O(n), where n- number of URLs, each URL is processed one after the other.

Parallel Executor with Process Pool Executor: O(n), where n- number of URLs. Processes run concurrently.

Parallel Executor with multiprocessing Pool: O(n), where n- number of URLs. all available CPU-cores are used.

Parallel Execution with threading (semaphore): O(n), where n- number of URLs. Threads run concurrently, limited by the semaphore = 5.

Space complexity:

Serial: O(1), 1 downloaded files stored in memory in a moment.

Parallel Executor with Process Pool Executor: O(n), where n- number of processes.

Parallel Executor with multiprocessing Pool: O(n), where n- number of processes.

Parallel Execution with threading (semaphore): O(n), where n- number of threads.

Test results and analysis:

Serial: 20.38 second(s). The longest time, the sum of each file downloaded.

Parallel Executor with Process Pool Executor: 6.45 second(s). Significantly faster due to concurrent processing.

Parallel Executor with multiprocessing Pool: 6.48 second(s). Significantly faster due to concurrent processing.

Parallel (Thread with semaphore): 8.65 second(s). Faster then serial and slower then Parallel (Process Pool Executor and Pool).

Based on the tests performed, the Parallel Executor method with Process Pool Executor was used to implement the project because it provides:

- The fastest execution time
- Maximum use of computing resources, utilizing all available processor cores.

- audio_extractor.py

Strategy: The script provides tests of multiple methods, including serial processing, multiprocessing, threading and asyncio.

Time complexity:

Serial: O(n), where n- number of video files. Each video file is processed one after the other.

Multiprocessing Pool: O(n), where n- number of video files. Processes run concurrently.

Threading: O(n), where n- number of threads. Threads run concurrently.

Asyncio: O(n), where n- number of video files. Asyncio run concurrently.

Space complexity:

Serial: O(1), 1 downloaded files stored in memory in a moment.

Multiprocessing Pool: O(n), where n- number of processes. Each process run concurrently for each file.

Threading: O(n), where n- number of threads. Threads run concurrently for each file.

Asyncio: O(n), where n- number of video files. Asyncio tasks run concurrently for each file.

Test results and analysis:

Serial: 22.00 second(s). The longest time.

Multiprocessing Pool: 20.99 second(s).

Threading: 17,32 second(s).

Asyncio: 16.15 second(s). Faster than other methods.

Based on the tests performed, the asynchronous programming method was used to implement the project because it provides:

- The fastest execution time
- Maximum use of computing resources.

- text_extractor.py

Strategy: The current implementation uses a serial approach to process each audio file one by one. This method is straightforward to implement and works well for a small number of files. However, for larger datasets (even in this task), parallel processing methods (such as threading or multiprocessing) could be used to speed up and improve efficiency.

Time complexity is O(1).

Space complexity is O(n), as extracted text files are stored in memory for further processing operations.

- emotion_extractor.py, sentiment_analysis.py, translator.py

Strategy: Each script sequentially reads and processes each text file to extract emotion information, sentiment analysis, or text translation. Files are initially limited in size, which determines the time complexity of the operation to be O(n), where n can represent the number of characters in the text, the number of tokens, or the number of sentences. The space complexity is estimated to be O(1), where 1 file is analysis results (for example, the size of emotional text, sentiment analysis results, or translated text).

Conclusions and recommendations

The project successfully implemented various multiprocessing, threading or asynchronous programming concepts to achieve the desired tasks. Each of these methods was applied to specific subtasks based on results of test performance outcomes.

Serial Processing: was used for simple and straightforward tasks.

Multiprocessing: was used for tasks such as video downloading. It can be applied for other process such as audio extraction and text extraction.

Threading: could have been to utilized in the project to implement the tasks of downloading video, extracting audio and text. However, the data processing time was slightly worse than that of Multiprocessing and Asynchronous programming.

Asynchronous programming: was found effective for audio extraction tasks, but could also be used for video processing and text extraction.

In general, with a small amount of data (as used in the project), any of the proposed methods can be suitable. However, for larger datasets, parallel processing techniques and asynchronous methods are preferable.