ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal
Regular Issue, Vol. 13 (2024), e31615
eISSN: 2255-2863
DOI: https://doi.org/10.14201/adcaij.31615

A Parallel Approach to Generate Sports Highlights from Match Videos Using Artificial Intelligence

Arjun Sivaraman, Tarun Kannuchamy, Anmol Anand, Shivam Dheer, Devansh Mishra, Narayanan Prasanth, and S. P. Raja

School of Computer Science and Engineering, Vellore Institute of Technology, Vellore, Tamil Nadu, India, 632014

✉ arjun.sivaraman2020@vitstudent.ac.in, tarun.kannuchamy2020@vitstudent.ac.in, anmol.anand2020@vitstudent.ac.in, shivam.dheer2020@vitstudent.ac.in, devansh.mishra2020@vitstudent.ac.in, nnprcd@gmail.com, avemariaraja@gmail.com

KEYWORDS

parallel computing; highlights generation; multiprocessing; speech to text; artificial intelligence; natural language processing; cosine similarity

ABSTRACT

Publishing highlights after a sports game is a common practice in the broadcast industry, providing viewers with a quick summary of the game and highlighting interesting events. However, the manual process of compiling all the clips into a single video can be time-consuming and cumbersome for video editors. Therefore, the development of an artificial intelligence (AI) model for sports highlight generation would significantly reduce the time and effort required to create these videos and improve the overall efficiency and accuracy of the process. This would benefit not only the broadcast industry but also sports fans who are looking for a quick and engaging way to catch up on the latest games. The objective of the paper is to develop an AI model that automates the process of sports highlight generation by taking a match video as input and returning the highlights of the game. The approach involves creating a list of words (wordnet) that indicate a highlight and comparing it with the commentary audio's transcript to find a similarity, making use of a speech-to-text conversion, followed by some pre-processing of the extracted text, vectorization and finally measurement of the cosine similarity metric between the text and the wordnet. However, this process can become time-consuming too, in case of longer match videos, as the computation times of the AI models become inefficient. So, we used a parallel processing technique to counter the time required by the AI models to compute the outputs on large match videos, which can decrease the overall time complexity and increase the overall throughput of the model.

1. Introduction

The broadcast industry has a mandatory practice of publishing highlights after a sports game. This is created to provide viewers with a quick summary of the game and highlight interesting events. However, the manual process of compiling all the clips into a single video can be time-consuming and cumbersome for video editors. Nowadays, Artificial Intelligence based models are used to develop and automate the process of sports highlight generation by taking the match video as input and returning the highlights of the game. Natural Language Processing (NLP) techniques enable the model to parse and process commentary data appropriately. The development of an AI model for sports highlight generation is a complex task that requires a deep understanding of both sports and technology. The model needs to be trained on a large dataset of sports videos to recognize and extract the most interesting events. The training process involves using supervised learning algorithms to classify the events into different categories, such as fall of wickets, exceptional fielding, and batsman hitting boundaries. The model then uses this classification to automatically identify similar events in future videos. The use of NLP techniques enhances the model's capabilities by allowing it to extract information from the commentary and other textual data associated with the video. This information can be used to further refine the model's classification of events and improve the accuracy of the generated highlights (Naik et al., 2022).

To optimize the overall efficiency of AI models, the parallelization of operations becomes necessary. Since AI models require intensive processing, parallelizing operations can help reduce the overall time taken and achieve maximum throughput. This approach enhances efficiency by distributing the workload across multiple computing units or processors (Wu, 2022; Paliwal et al., 2022; Malik et al., 2020). By parallelizing operations, the AI model can effectively handle the processing requirements involved in sports highlight generation. This approach proposes an AI model that automates the process of sports highlight generation by taking a match video as input and returning the highlights of the game. The model has been trained to recognize and extract the most interesting events from the video and provide a summary of the game that is both informative and engaging. The expected output data feed contains start/end time stamps of interesting clips from the given video feed.

In addition, we have addressed the current lack of parallel computing utilization in highlight generation methods. Our implementation aims to enhance efficiency within this domain, recognizing the potential for significant performance improvements, and hereby reducing the computation time with respect to the serial processing of the models, even on longer match videos.

2. Related Work

There is a myriad of approaches available to generate highlights from match videos. Shuka et al. (2018) proposed a model that made use of both event-based and excitement-based features to recognize and clip important events in a cricket match. These cues included replays, audio intensities, playfield scenarios, etc. In the said approach, the video was segmented, and each segment was passed through a Convolutional Neural Network (CNN) + Support Vector Machine (SVM) framework, which identifies the segment as a replay or advertisement or a part of the game. Further, using a scoreboard detection framework which employed an Optical Character Recognition (OCR) classifier, events during the game were highlighted and using the CNN + SVM framework mentioned before, playfield scenarios were identified. Cues in the audio, such as excitement in the crowd, were also analyzed for various other miscellaneous events. A combination of all these features was used to identify boundaries,

wickets, and milestone celebrations. Islam et al. (2018) proposed a novel method for video summarization in sports videos, specifically soccer matches. While visual features were commonly used, audio features were also important for capturing exciting events. The proposed approach focused on extracting audio features using Empirical Mode Decomposition (EMD) to filter out unwanted noises, such as audience sounds. By decomposing the audio signal into different frequency bands, relevant features were extracted. The method was evaluated on real soccer videos, and the results showed that it achieved 96 % accuracy in detecting goal events, surpassing the state-of-the-art method, available at that time, by 42 %. This demonstrates the effectiveness of incorporating audio features for improved video summarization in sports videos. Midhu et al. (2018) proposed an approach for producing 15 cricket video highlights by identifying significant occurrences. There were two components to the process. Based on the difference in the hue histogram, critical frames were located in the first section. The lack of a scoreboard was then used to categorize frames as replay or real-time frames. Based on the Dominant Grass Pixel Ratio, real-time frames were further divided into field view and non-field view classes. Edge detection was used to categorize on-field footage into pitch view and boundary view, while close-up and crowd frames were identified individually. The Apriori method was used in the second section to perform idea mining on labelled frame events. This kind of an approach cannot detect events which are independent of a scoreboard such as close calls, DRS reviews, any exceptional field play which is not a boundary or a wicket. Trivedi et al. (2018) described an approach of speech recognition via pattern matching in their study, which removed the errors that occurred due to segmentation or classification of smaller acoustically variable units which are dependent on the speaker. It untangled speech-to-text conversion, confirming fast convergence and enhanced recognition accuracy. There was limited diversity in datasets that were used in the study which may have not correctly represented the bigger population or captured the full range of variations in real-world datasets. Introduction of bias and impact on the validity of the results may be caused due to the authors' preference.

Wei and Zou (2019) proposed four easy data augmentation techniques that are explained below. Synonym Replacement involved substituting words that were there in the text with their synonyms to generate different versions of the text. The process of adding randomly chosen words from the vocabulary to the text to create new variations is called Random Insertion. Random Swap entails exchanging two randomly selected words in the text to produce different versions. Random Deletion, as the name suggests, deletes randomly chosen words from the text to create divergent variations. All these techniques aim to generate additional training data by augmenting the existing data with semantically similar text. Here the performance of these techniques was evaluated on six different datasets and the result showed that these techniques can significantly improve the performance of text classification model. The study by Iqbal et al. (2020) provided an overview and differentiation of various feature selection methods including filters, wrappers, and embedded methods. Feature selection is an essential step in classification of the text. Different methods have different impacts on text classification performance. The effectiveness of the various feature selection methods varies with the type and size of the dataset and the classification algorithm used. Due to the complex nature of the text data, characterized by the very high dimensionality and sparsity, there may be many challenges in feature selection such as identifying the most pertinent features for a particular text classification problem which is a difficult task and demands expertise.

Gagliardi et al. (2020) outlined the creation of an NLP pipeline comprising multiple stages. Initially audio files were preprocessed to remove noise and convert them into text. The next stage involved aligning the text with the corresponding audio, which can be challenging due to speech variability. The authors proposed several techniques to enhance alignment accuracy, incorporating acoustic

features and machine learning algorithms. In the last stage, various NLP techniques were applied to the transcribed text, including part-of-speech tagging, sentiment analysis, and named entity recognition. Limitations: Speech therapy often includes the analysis of speech recordings, which can be time consuming and challenging for therapists. The authors suggested that an NLP pipeline could be used to automate parts of this process, making it easier and faster for therapists to transcribe and analyze speech recording. Thompson et al. (2015) analyzed the performance of eight similarity measures, including cosine similarity, Jaccard similarity and Euclidean distance on two datasets. The first dataset consists of similar text pairs, while the second dataset consists of dissimilar text pairs. Evaluation metrics such as precision, recall and F1-score were employed to compare the performance of the similarity measures on both datasets. Additionally, the authors have analyzed how the measures respond to text length variations by comparing their performance on subsets of the datasets with various text lengths. There are two primary challenges highlighted by the authors in their work. First, the absence of a standardized evaluation methodology for text similarity measures makes it challenging to compare and replicate results across different studies. Second is that the selection of a suitable dataset for evaluations poses a challenge since the performance of similarity measures can vary depending on the dataset's type and size. The study by Gunawan et al. (2018) used cosine similarity as a method to determine the textual relevance between two documents. Cosine similarity is a measure of similarity that calculates the cosine of the angle between two non-zero vectors in an inner product space. Here, the documents were represented as vectors, with each element representing a word or phrase within the documents. Cosine similarity between these vectors was computed to ascertain the similarity between the documents. The effectiveness of cosine similarity in calculating text relevance relies heavily on the quality of the text data, encompassing the accuracy of document representation and the preprocessing steps. When dealing with large datasets or multiple documents, cosine similarity may become computationally demanding. The presence of noise or outliers in the data can affect the accuracy of the cosine similarity for text relevance calculation. Here, the author does not explicitly discuss approaches to tackling these challenges.

The study by Alzahrani (2016) emphasized the utilization of parallel programming techniques to improve the efficiency of NLP in analyzing large-scale data. The authors acknowledged that the exponential growth of digital data has resulted in a substantial volume of text data requiring processing, surpassing the capabilities of traditional NLP methods. Addressing this issue, the author proposed that parallel programming approaches can be employed to distribute the processing workload across multiple processors or nodes, thus enabling the effective handling of the data volume. The author also compared the efficiency and speedup achieved by the parallelization of various natural language processing methods to their sequential executions. Comparisons were made based on the method, the number of cores present in the computer (8-core, 16-core and 32-core) and the type of programming tool used (Python, MATLAB and C#.NET) on different dataset sizes. The methods compared included those of tokenization, stemming and lemmatization, POS tagging, near duplicate detection and document ranking using TF-IDF. Of these extensive comparisons, the comparisons of our interest include the methods of tokenization, lemmatization and TF-IDF weighing and cosine similarity, carried out on an 8-core computer and programmed using Python. For word unigrams and 3-grams tokenization, the parallel execution on an 8-core computer was 2 times faster than the sequential execution. However, for sentence and 5-gram tokenization, the sequential approach was faster than the parallel one. In case of stemming, overall, the parallelized algorithms executed faster than their sequential counterparts. Finally, document ranking using cosine similarity, which made use of a TF-IDF vectorizer performed 2 times faster than its sequential counterpart on an 8-core computer, a speedup which was regarded

as remarkable by the author. Upon running the same methods on different sizes of data, the author observed that the speedup achieved through parallelization depended upon the complexity of the NLP algorithm, the size of the dataset and the number of cores. By keeping the number of cores constant, the study highlighted that the effect of parallelization is not evident, if the algorithm or process is simple and basic, even if the size of the data is big, and thus in some cases, the serial executions outperformed the parallel ones.

3. Proposed Work

To improve the limitations discussed by Shukla et al. (2018), Islam et al. (2019) and Midhu et al. (2018), we have used a commentary transcript analysis-based approach and have applied parallelism in its operations (Alzahrani, 2016). Python has been used as the programming language for the development of the proposed work, as it offers thread and process-level parallelism via its multiprocessing module (Aziz et al., 2021) and supports many libraries for natural language processing and data analysis. The process of automatically generating highlights from a sports game video, uploaded by a user, involves several steps and is depicted in Figure 1. The proposed methodology focuses on data parallelism i.e., Single Instruction Multiple Data (SIMD). In the case of SIMD, we would be performing the same operations on multiple fragments of the overall data across different threads. This approach is shown inside the boxed region in Figure 1.

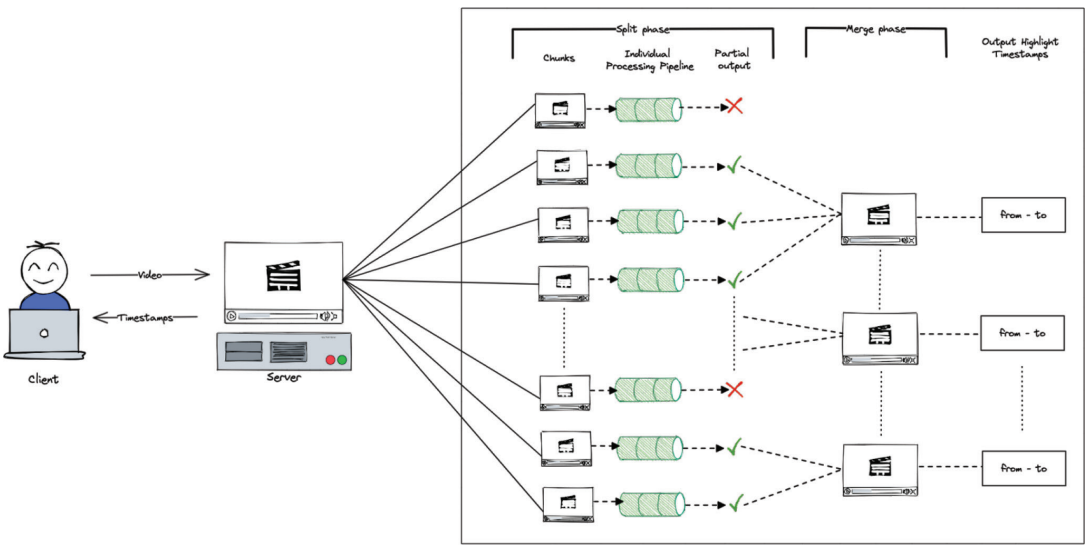

Figure 1. Architecture diagram for generating highlights from a game

The parallelization algorithm is divided into four tasks: speech-to-text conversion, NLP preprocessing pipeline, vectorization and similarity checking and finally the merging phase. Figure 2 shows the levels of parallelization in the proposed algorithm. Mutexes are used to create critical sections at places where

simultaneous writes take place in the shared memory variables, in order to avoid race conditions. The identification and avoidance methods for race conditions are discussed further in their respective phases.

The first step towards preparing the data for parallelization involves converting the video into audio and then breaking down the obtained audio into smaller bits using the pydub library. Pydub is a simple and easy-to-use audio processing library for Python. The number of chunks is decided on the basis of the number of threads spawned by the system. Each thread is assigned a single or a group of chunks to process and carry out the operations discussed below.

3.1. Speech-to-Text Conversion

From this point on, all the chunks are processed in parallel by threads up to the merge phase by applying the same steps of processing on each of them. It begins by transcribing the text from the commentary audio using the OpenAI's Whisper base model (an open-source speech-to-text model based on Radford et al. (2022) on each chunk of audio. One of the places where race conditions can occur is during the addition of the extracted text segments to the global segments array, as in line 19 of Algorithm 1. To avoid this, we define a lock variable «sl» of the type of a mutex, which is used to prevent multiple threads appending to the global segments array at the same time. Only one thread can acquire the «sl» lock at a time, to enter the critical section for appending to the array, and then releases the lock for other competing threads to acquire it, thus avoiding race conditions at this point.

Algorithm 1: Text extraction from audio sub clip (within each thread)

```
1.  import whisper
2.  model = whisper.load_model("base")
3.  def extract_text(sl: Lock, segments, audio, start, end):
4.    # segments is the shared global array accessed by all threads to store the transcribed text
5.    global model # The OpenAI Whisper speech-to-text model
6.    try:
7.      # Save the sub clip audio
8.      path = f"{AUDIO_DIR}{str(int(start + end))}.wav"
9.      audio[start*1000: end*1000].export(path, format="wav")
10.     # Transcribe audio
11.     output = {"text": "", "segments": []}
12.     try:
13.       output = model.transcribe(path)
14.     finally:
15.       # Add segments to the thread safe segment list
16.       for i in output["segments"]:
17.         sl.acquire()
18.         try:
19.           segments.append({"text": i["text"], "start": i["start"] + start, "end": i["end"] + start})
20.         finally:
21.           sl.release()
22.   finally:
      return
```

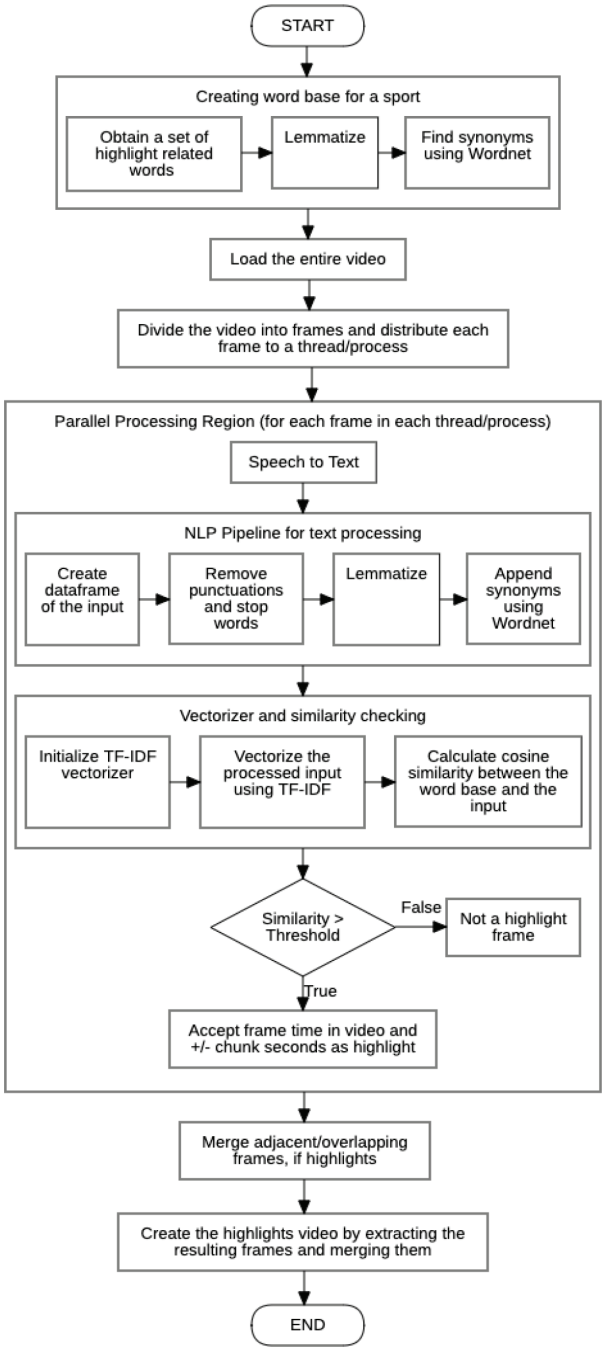

Figure 2: Flow diagram of the proposed work

3.2. Pre-Processing NLP Pipeline

The text extracted from each chunk goes through an NLP pipeline where it undergoes several processes such as conversion to lowercase, removal of punctuation, removal of repeated words, stemming, and lemmatization (Gagliardi et al., 2020). This helps to standardize the text and reduce noise. The first step in NLP text preprocessing is tokenization, where the text is divided into smaller units called tokens. These tokens can be words, phrases, or even individual characters. Tokenization helps to break down the text into more manageable pieces for analysis. The text is then converted to lowercase, which helps to eliminate any discrepancies that may occur due to capitalization. Next, punctuation marks such as periods, commas, and exclamation marks are removed from the text. This helps to ensure that the text is consistent, and that punctuation does not interfere with the analysis. Stop words are common words such as «the», «a», and «an» that are frequently used in a language but do not add any value to the meaning of the text. Removing these words helps to reduce noise in the text and makes it easier to identify the relevant keywords and phrases. Stemming and lemmatization are also applied to the text. Stemming involves reducing words to their root form, while lemmatization involves reducing words to their base form. This step helps to group together words with similar meanings and reduces the overall number of words in the text.

3.3. Vectorization and Similarity Checking

A wordnet consisting of common cricket highlight words is created and stored for future use. The wordnet is a database of words that are related to each other based on their meaning. It is used to identify words that are relevant to the highlights of the game. The wordnet and the text in each chunk goes through the TF-IDF vectorizer provided by the scikit-learn library. This vectorizes the text and prepares it in a form that can be comparable using a similarity metric. Based on the discussions in Thompson et al. (2015), in our case we have used cosine similarity. The cosine similarity checks, as in Gunawan et al. (2018), is performed between the wordnet and the chunk text to determine if there is any similarity between the received text and the set of words indicating a highlight. The threshold value was determined by running the algorithm on sample transcripts generated from a variety of audios. On the basis of this experiment, we concluded to use a threshold value based on factors such as audio quality and clarity in commentary (the lesser the background noise by crowd, the better the clarity). Table 1 shows the threshold values, determined on the basis of these factors for different scenarios.

Table 1. Threshold values in different scenarios

Scenario	**Threshold Value**
High audio quality with clarity:	0.06
High audio quality with crowd noise:	0.03
Low audio quality with clarity:	0.025
Low audio quality with crowd noise:	0.01

If there is a significant similarity that passes the set threshold value, the timestamps of the obtained highlights are returned to the server. There is a possibility of race conditions occurring in the part where

all the threads append their timestamps to the global results array at line 32 of Algorithm 2. Thus, we make use of a lock variable «rl» to create a mutex for entering the critical section of the appending of the global result array. This implementation can be found from lines 30 to 34 of Algorithm 2. Finally, these highlights are generated by taking some time before and after the highlight to ensure that the complete highlight is captured.

Algorithm 2: Vectorization and similarity checking

```
1.  import nltk
2.  from nltk.corpus import wordnet
3.  import pandas as pd
4.  from sklearn.feature_extraction.text import TfidfVectorizer
5.  from sklearn.metrics.pairwise import cosine_similarity
6.  nltk.download('wordnet')
7.  nltk.download('omw-1.4')
8.  positive = list()
9.  # Words indicating highlights (in case of cricket)
10. add = [ "beauty", "fifty", "century", "perfect", "magnificient", "batting", "fielding", "bowl-
    ing", "catch", "out", "stumped", "bowled", "night", "wicket", "review", "DRS", "cuts", "out",
    "short", …]
11. for i in add:
12.   for synset in wordnet.synsets(i):
13.     for lemma in synset.lemmas():
14.       positive.append(lemma.name())
15. strings = ' '.join(positive)
16. def calculate_similarity(l: Lock, result, segment):
17.   d = [segment["text"], strings]
18.   # Vectorize the strings
19.   Tfidf_vect = TfidfVectorizer()
20.   vector_matrix = Tfidf_vect.fit_transform(d)
21.   tokens = Tfidf_vect.get_feature_names_out()
22.   # Calculate cosine similarity score
23.   cosine_similarity_matrix = cosine_similarity(vector_matrix)
24.   # Create a dataframe from the cosine similarity matrix using the phrases as tokens
25.   doc_names = [f'doc_{i+1}' for i, _ in enumerate(cosine_similarity_matrix)]
26.   r = pd.DataFrame(data= cosine_similarity_matrix, index=doc_names, columns=['Phrase',
    'Strings'])
27.   score = r['Phrase'].values[1]
28.   # Accept as highlight if greater than threshold
29.   if (score >= 0.02500000000000):
30.     l.acquire()
31.     try:
32.       result.append([segment["start"], segment["end"]])
33.     finally:
    1.release()
```

3.4. Merging

At this phase, the timestamps from all the threads are merged into a single list of timestamps. This step waits for all the threads to finish processing and appending their outputs before moving ahead. Computation is carried out serially from this point. The obtained timestamps are sorted in order. Then, adjacent or overlapping timestamps are combined with respect to their start and end timestamps into a single highlight timestamp and added to the final list of highlights. The final step is extracting the parts of the original video corresponding to each of the timestamps and merging all of them to form the final highlights output video. This compilation is done by using the moviepy module's extract_subclip and concatenate_videoclips methods. This process saves a lot of time and effort for video editors and broadcasters who would otherwise have to manually identify and compile interesting clips from a sports game video. It also helps to ensure that all the important highlights are captured, improving the overall quality of the highlights reel.

Algorithm 3: Parallel programming approach using Python multiprocessing module

```
from multiprocessing.pool import ThreadPool
from multiprocessing import Lock
from moviepy.video.io.ffmpeg_tools import ffmpeg_extract_subclip
from moviepy.editor import VideoFileClip, concatenate_videoclips
from pydub import AudioSegment
if __name__ == "__main__":
  # Get the video duration
  vid = VideoFileClip(SAMPLES_DIR + VIDEO_NAME)
  duration = vid.duration
  # Convert to audio
  video = AudioSegment.from_file(SAMPLES_DIR+VIDEO_NAME, format="mp4")
  audio = video.set_channels(1).set_frame_rate(16000).set_sample_width(2)
  audio.export(SAMPLES_DIR+AUDIO_NAME, format="wav")
  audio = AudioSegment.from_file(SAMPLES_DIR+AUDIO_NAME, format="wav")
  try:
    segments = list()
    # Create a lock to append to the segments extracted
    sl = Lock()
    # Create the thread pool for extraction
    with ThreadPool() as pool:
      # No. of processes
      np = pool._processes
      # Chunk size
      chunk = duration / np
      # Excess time for overlaps
      excess = 3
      # Create partitions by taking excess
      args = [[chunk * (i) - excess, chunk * (i + 1) + excess] for i in range(np)]
      args[0][0] += excess
```

```
        args[-1][1] -= excess
        pool.starmap(extract_text, [tuple([sl, segments, audio]) + tuple(i) for i in args])
    result = list()
    # Create a lock to append to the result
    rl = Lock()
    # Create the thread pool for similarity check
    with ThreadPool() as pool:
      # Chunk size calculated dynamically
      pool.starmap(calculate_similarity, [tuple([rl, result, i]) for i in segments])
    result = sorted(result)
    # Merge adjacent results
    if len(result) > 0:
      res = [[result[0][0] - 8 if result[0][0] - 8 > 0 else 0, result[0][1] + 6]]
      for i in range(1, len(result)):
        if res[-1][1] + 6 >= result[i][0] - 6:
          if res[-1][1] + 6  < result[i][1] + 6:
            res[-1][1] = result[i][1] + 6
        else:
          res.append([result[0] - 6 if result[0] - 6 > 0 else 0, result[i][1] + 6])
      # Extract sub clip from res[i][0] to res[i][1] using ffmpeg_extract_subclip into ETC_DIR
      # Concatenate highlights from sub clips in ETC_DIR using concatenate_videoclips
  finally:
    # Cleanup and delete temporarily created files
```

4. Results and Discussion

To test our algorithm, we took the «2007's India v/s Australia» match (https://youtu.be/wDQ4l9EiIfg) and took out a 30-minute extract from the same. In our tests, the algorithm was set to use 8 processes, which was the default value. Table 2 gives the video chunk division amongst the processes:

Table 2. Division of input video into chunks (in seconds)

Chunk	Start of chunk	End of chunk
1	0.0000	37.0325
2	27.0325	69.0650
3	59.0650	101.0975
4	91.0975	133.1300
5	123.1300	165.1625
6	155.1625	197.1950
7	187.1950	229.2275
8	219.2275	256.2600

Each of the chunks was passed through the NLP pipelines and all the probable highlights were inserted into an array. On merging the adjacent timestamps, Table 3 shows the final output array that was obtained, which were then merged to create the final highlights video:

Table 3. Final timestamps (in seconds)

Timestamp	Start time	End time
0	10.0	20.0
1	44.0	54.0
2	58.0	76.0
3	80.0	90.0
4	97.0	109.0
…	…	…
51	1746.38	1754.38
52	1769.38	1783.38
53	1804.38	1814.38

Table 4 below gives an analysis of the outputs of the algorithm with respect to the input given:

Table 4. Results and comparisons

Metric	Value
Time taken in serial execution:	161 seconds
Time taken in parallel execution:	58 seconds
No. of probable highlights present in input:	49
No. of correct timestamps identified as highlights in output:	46
No. of incorrect timestamps identified as highlights in output:	7
No. of missed highlights in output with respect to input:	5

The findings of this study highlight the substantial potential of parallelizing Natural Language Processing (NLP) for automating highlight generation in the broadcast industry. Our algorithm demonstrated significant efficiency improvements through parallel execution, completing the sample task in just 58 seconds. This represents an approximate 2.8-fold increase in speed compared to the 161 seconds required for serial execution. This substantial speedup underscores the significant advantage of employing parallel processing in our approach, especially in scenarios demanding real-time or near-real-time broadcast.

Around 8 to 10 seconds of time was lost to audio splitting costs (apart from other parallelization costs) in case of the parallel algorithm. Despite this overhead, the parallel approach is still faster than the sequential approach. Utilizing an 8-core parallel programming setup, speech-to-text processing in parallel took 15.1 seconds was approximately 9 times faster compared to the serial approach which took 139.2 seconds for the same phase. In case of the pre-processing pipeline consisting of tasks such

as tokenization, conversion to lowercase, punctuation removal, elimination of repeated words, stemming, and lemmatization, the results for the serial approach proved to be just equal or even faster in some cases with a time of 0.9 seconds, compared to the parallel method which took 1.4 seconds. At the final level of parallelization, the vectorization using TF-IDF and cosine similarity checking, the serial vectorization process took 1.9 seconds, whereas in parallel, it was completed in 1.2 seconds, resulting in approximately a 1.5 times speedup. The merging phase common to both approaches took around 10.4 seconds to execute.

The use of the SIMD model in our case, reveals an important caveat. Parallel processing may not be efficient for use in algorithms which are simple or basic, even with the increasing size of data. This can be seen in the results from the pre-processing phase, wherein surprisingly, sequential preprocessing was faster than its parallel counterpart. This can be attributed to the fact that parallelism introduces overhead, including lock creation and thread and core management, which can extend processing times and these costs maybe greater than the costs of serial execution if the process is very basic or simple. The algorithms of tokenization, punctuation and stop words removal, stemming and lemmatization were not complex enough for the realization of these parallelization costs. We can say that our parallelism's advantages are most apparent when tasks on the data demand higher complexity for each iteration as in the cases of speech-to-text conversion and vectorization and cosine similarity checking and proved to be almost ineffective in case of the pre-processing pipeline.

5. Conclusion

The use of NLP enabled methods to automatically generate highlights from an input video file can significantly improve the speed and efficiency of the process of producing highlights videos in the broadcast industry. The application proposed in this study can help video editors save time and effort by automatically identifying potential areas for highlights and compiling all the clips into a single compiled video. This technology is a significant development in the broadcast industry and has the potential to change the way highlights are generated and consumed. Sections of the algorithm where the parallelization proved to be efficient are the automatic speech recognition to extract text and the vectorization and similarity checking procedures. However, one of the key challenges in developing an AI model for sports highlight generation is the need to balance between the quantity and quality of the generated highlights. The model needs to be trained to recognize and extract the most interesting events while avoiding generating too many irrelevant or redundant clips. This requires a careful tuning of the model's parameters and the use of advanced algorithms such as deep learning and reinforcement learning. The model can be further extended to align generated highlights with the expectations of the viewers, such as showing the clips of their favorite teams or players more in the spotlight or narrowing down to more spectacular events during the game. This requires understanding the viewers' preferences and expectations and incorporating this information into the model's training process.

6. Appendices

6.1. Appendix A – List of Abbreviations

A.1. AI – Artificial Intelligence.
A.2. NLP – Natural Language Processing.

A.3. SIMD – Single Instruction Multiple Data.
A.4. ASR – Automatic Speech Recognition.
A.5. TF-IDF – Term Frequency-Inverse Document Frequency.
A.6. SAMPLES_DIR – Base directory where the input video file is stored.
A.7. AUDIO_DIR – Directory to store the audio sub clips used for processing by each thread.
A.8. ETC_DIR – Directory to store the extracted video sub clips when compiling the highlights video.
A.9. OUT_DIR – Directory to store the output of the algorithm i.e., the compiled highlights video.
A.10. VIDEO_NAME – Name of the input video file.
A.11. AUDIO_NAME – Name of the converted audio file which is to be processed.

6.2. Appendix B – Technical Specifications

B.1. Python: Python is a high-level, interpreted programming language that is widely used for web development, data analysis, scientific computing, and artificial intelligence applications.
B.2. Python Multiprocessing Module: In Python, the multiprocessing module includes a very simple and intuitive API for dividing work between multiple processes. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads.
B.3. NLTK: NLTK (Natural Language Toolkit) is a popular open-source library for working with human language data in Python. It provides a suite of libraries and programs for natural language processing (NLP) tasks such as tokenization, stemming, tagging, parsing, and sentiment analysis.
B.4. OpenAI Whisper: OpenAI's Whisper is a state-of-the-art speech synthesis model that can generate high-quality, human-like speech in real-time. It uses a neural network-based approach to convert text input into speech output.
B.5. Pydub: Pydub is a simple and easy-to-use audio processing library for Python. It allows users to manipulate audio files, including cutting, concatenating, and applying various audio effects.
B.6. Moviepy: MoviePy is a Python library for video editing that provides a range of functions for creating, modifying, and combining video clips. It can be used to add text, images, and audio to videos, and to apply various effects and transitions.
B.7. Scikit-learn: Scikit-learn is a machine learning library in Python that provides a range of supervised and unsupervised learning algorithms for classification, regression, clustering, and dimensionality reduction. It is built on top of NumPy, SciPy, and matplotlib.

6.3. Appendix C – List of Figures

6.4. Appendix D – List of Tables

6.5. Appendix E – List of Algorithms

Algorithm E.1. Text extraction from audio sub clip (within each thread).
Algorithm E.2. Vectorization and similarity checking.
Algorithm E.3. Parallel programming approach using Python multiprocessing module.

References

Alzahrani, S. M. (2016). Parallel programming approaches for efficient natural language processing of big data. *International Journal of Computer Science and Information Security (IJCSIS)*, *14*(10).

Aziz, A. Z., Abdulqader, D. N., Sallow, A. B., & Omer, H. K. (2021). Python Parallel Processing and Multiprocessing: A Review. *Academic Journal of Nawroz University*. *10*(3), 345–354. https://doi.org/10.25007/ajnu.v10n3a1145

Gagliardi, G., Gregori, L., & Ravelli, A. A. (2020). An NLP Pipeline as Assisted Transcription Tool for Speech Therapists. *12th International Conference on Language Resources and Evaluation (LREC 2020)*, 124-130.

Gunawan, D., Sembiring, C. A., & Budiman, M. A. (2018). The Implementation of Cosine Similarity to Calculate Text Relevance between Two Documents. *Journal of physics: Conference Series*, *978*(1), 012120. https://doi.org/10.1088/1742-6596/978/1/012120

Iqbal, M., Abid, M. M., Khalid, M. N., & Manzoor, A. (2020). Review of feature selection methods for text classification. *International Journal of Advanced Computer Research, 10*(49), 138-152. https://doi.org/10.19101/IJACR.2020.1048037

Islam, M. R., Paul, M., Antolovich, M., & Kabir, A. (2019). Sports Highlights Generation using Decomposed Audio Information. *2019 IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*, Shanghai, China, 2019. 579-584. https://doi.org/10.1109/ICMEW.2019.00105

Malik, M., Malhotra, S., & Prasanth, N. (2020). Time Improvement of Smith-Waterman Algorithm Using OpenMP and SIMD. *Communications in Computer and Information Science*. *1206*(CCIS), 686–697. https://doi.org/10.1007/978-981-15-4451-4_54

Midhu, K., & Padmanabhan, N. K. A. (2018). Highlight generation of cricket match using deep learning. In *Lecture Notes in Computational Vision and Biomechanics* (Vol. 28, pp. 925-936). Springer, Cham. https://doi.org/10.1007/978-3-319-71767-8_79

Naik, B.T., Hashmi, M.F., & Bokde, N.D. (2022). A Comprehensive Review of Computer Vision in Sports: Open Issues, Future Trends and Research Directions. *Applied Sciences, 12*(9), 4429. https://doi.org/10.3390/app12094429

Paliwal, M., Chilla, R., Prasanth, N., Goundar S., & Raja S.P. (2022). Parallel implementation of solving linear equations using OpenMP. *International Journal of Information Technology*, *14*(2), 1677–1687. https://doi.org/10.1007/s41870-022-00899-9

Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., & Sutskever, I. (2022). Robust Speech Recognition via Large-Scale Weak Supervision. *40th International Conference on Machine Learning, PMLR*, *202*, 28492-28518.

Shukla, P. et al. (2018). Automatic Cricket Highlight Generation Using Event-Driven and Excitement-Based Features. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Salt Lake City, UT, USA, 2018. 1881-18818. https://doi.org/10.1109/CVPRW.2018.00233

Thompson, V. U., Panchev, C., & Oakes, M. (2015). Performance Evaluation of Similarity Measures on Similar and Dissimilar Text Retrieval. *2015 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K)*, *1*, 577-584. https://doi.org/10.5220/0005619105770584

Trivedi, A., Pant, N., Shah, P., Sonik, S., & Agrawal, S. (2018). Speech to text and text to speech recognition systems - A Review. *IOSR Journal of Computer Engineering*, *20*(2), 36-43.

Wei, J., & Zou, K. (2019). EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. *2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China, 2019, 6382–6388. https://doi.org/10.18653/v1/D19-1670

Wu, L. (2022). Research on the Development and Application of Parallel Programming Technology in Heterogeneous Systems. *Journal of Physics: Conference Series*. *2173*(1), 012042. https://doi.org/10.1088/1742-6596/2173/1/012042