AutoVideoEdit: Development and Performance of Inner-Video-Search Python Package

Brendan Mitchell, Hyuntae Park, Ganesh Pimpale

---

0. Abstract

AutoVideoEdit (AVE) is a Python package designed to automate particularly tedious tasks involved with video editing. Specifically, this paper will focus on the development and performance of the video clip search by visual description feature. AVE allows users to easily load their videos and splice a series of queries together into a final edit within ten lines of code. We present an overview of AVE's internal design, performance, strengths, limitations, and future developments.

---

1. Motivation

Video editing is a laborious process that can take hours. Professional recommendations cite that every minute of finalized content should take between 30 minutes to an hour to edit. This problem continues to scale as a video editor is provided with more raw data. AutoVideoEdit (AVE) is designed to be a user-friendly Python package that automates the majority of highly time-consuming and tedious tasks. Such tasks include but are not limited to: searching for parts of videos, syncing audio and video files, and syncing video to music.

Eventually, AVE will encapsulate all these features and more. However, as of now, the software is only capable of inner-video-search. While current software to specifically search for parts of a video exists, they are paid platforms, closed source, and not designed for the process of video production, and therefore difficult to integrate into editing pipelines.

As a Python package, AVE is designed to be multi-functional, such as a base layer for a more complex application, a preprocessing step for large amounts of video data, a tool to edit videos in command line applications like iPython, or any new use a user may find.

2. Background Research

Computer vision is a rapidly growing field where deep machine learning approaches have gained a good reputation for extracting informative features from images; however, these complex approaches are reliant on domain knowledge, limited to a narrow set of features and categories, manually labeled, and computationally costly.

OpenAI's Contrastive Language-Image Pre-Training (CLIP) aims to mitigate these limitations. CLIP is a neural network trained on 400 million image and text pairs from the internet, integrating zero-shot transfer, natural language supervision, and multimodal learning methods as described in detail elsewhere [Radford et al. 2021]. In brief, CLIP first learns to predict the text that matches an image, then leverages those trained image-text representations to connect to and learn from a natural language which enables zero-shot transfer for unseen classes (i.e., flexible and generalized learning). Specifically, CLIP computes the cosine similarity of embedded features in text and images, scales by a temperature parameter, and normalizes it into a probability distribution via a softmax function. The prediction layer is a multinomial logistic regression classifier with L2-normalized inputs and weights, no bias, and temperature scaling. This model generates weights of a linear classifier based on the text that specifies image representations for each class of image-text pairs to predict the most probable image-text pair. This classifier can then be used for all subsequent predictions. CLIP outperforms traditional linear classifiers (e.g., ImageNet and ResNet) on zero-shot transfer across 20 datasets with a minimum of 16 examples per class [Radford et al. 2021].

PySceneDetect is a Python library that utilizes OpenCV to analyze a video to detect changes in scenes and split the video into scenes automatically [Castellano 2014-2022]. The detection method used is 'Content-Aware Detector', which detects jump cuts (i.e., a cut in film editing in which a single continuous sequential shot of a subject is broken into two parts) in the input video. It does so by finding areas where the difference between two subsequent frames exceeds the given threshold value. In detail, the content-aware scene detector initially converts the colorspace of each decoded frame from RGB into HSV, then it takes the average difference across all channels of the frames. If this exceeds a given threshold, a scene change is triggered.

By combining the ability to split videos into scenes with PySceneDetect and encoding keywords into these scenes with CLIP, we aim to design a program, termed AVE, that does the following: (1) splits a full-length video into scenes, (2) sample the frames in each scene, (3) encodes keywords into the video, and (4) successfully identifies the scene in a video that corresponds to any given keyword or phrase.

3. Design

3.1. Internal Structure

As discussed above, AutoVideoEdit is created by combining the functionality of multiple technologies, namely, CLIP and PySceneDetect. This process is illustrated in Figure 1.

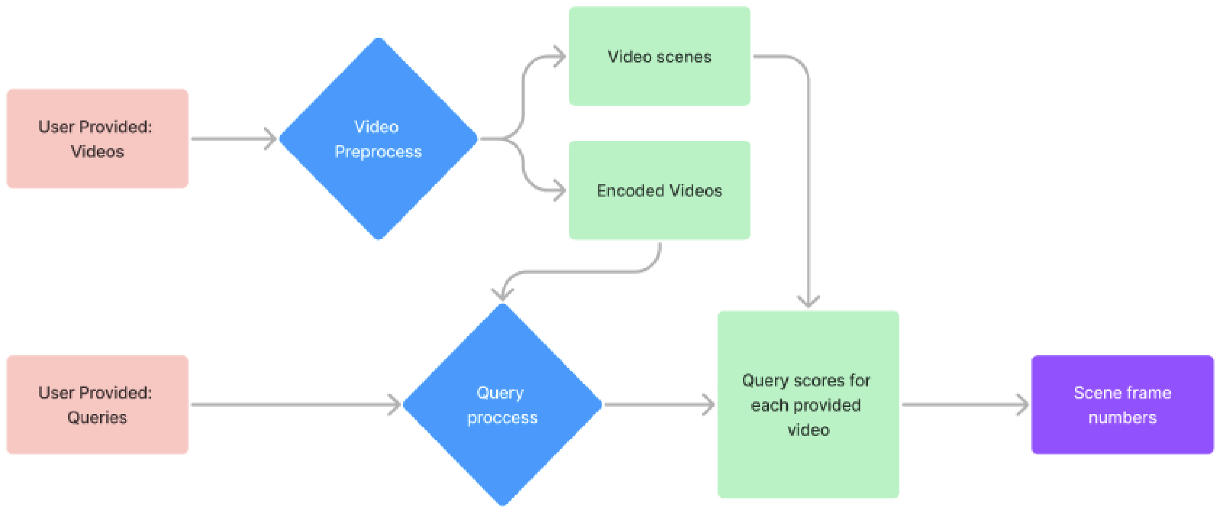

Figure 1. Summary of the AVE inner-video-search pipeline.

User-provided data consist of video data and a series of queries. Videos are processed before the queries. This is the most time-consuming process in the pipeline that uses the ContentDetect feature of PySceneDetect to divide the video up into a set of scenes. Additionally, this process also encodes the video, that is, every frame is tagged with keywords as found by passing the frame through CLIP. This process is done only once for each video and processing time is proportional to the number of frames.

Once all the videos have been processed, the queries will be processed. First, the query will be tokenized using CLIP’s language model. During this process, the language model will turn the string into a tensor. This tensor will be compared against the tensor generated for each frame and the similarity between the two tensors will be computed using the cosine similarity. Each query will be compared against each video such that the total number of run-throughs will be equal to the product of the number of videos and the number of queries. Although queries have to be processed more times than videos, each cycle is much shorter than preprocessing video data. However, it is important to note that this stage relies heavily on a dedicated GPU while video preprocessing is memory and CPU intensive.

Finally, the frame that has the maximum similarity to the query is selected and the clip that contains the frame is added to a queue that will be compiled into a video. In this prototype of AutoVideoEdit, the clips are directly joined to one another without any type of transition to create the final video.

3.2. Installation and Usage

In order to avoid version control issues between the technologies in use, Python 3.7 is recommended. Additionally, a Conda environment is also heavily recommended. After installing git and cloning the AutoVideoEdit repository, the setup process is as follows:

Create a Conda environment with Python 3.7:

```
$ cd AutoVideoEdit
$ conda create -n AutoVideoEdit python=3.7
$ conda activate AutoVideoEdit
```

Install CLIP and its dependencies:

```
$ conda install --yes -c pytorch pytorch=1.7.1 torchvision cudatoolkit=11.0
$ pip install ftfy regex tqdm
$ pip install git+https://github.com/openai/CLIP.git
```

Install the rest of the AVE's dependencies:

```
$ pip install numpy opencv-python scenedetect pytesseract
```

To run the example code:

```
$ python edit.py
```

AutoVideoEdit is designed to be used as a Python package. To demonstrate this functionality, we can look at the code in *edit.py*:

```
from AutoVideoEdit import AVE
vid1 = AVE()

vid1.addVideo('./test_vids/random_cat.mp4')
vid1.newQuery('petting a cat')
vid1.addVideo('./test_vids/baby_penguin.mp4')
vid1.newQuery('a small white penguin')
vid1.newQuery('a penguin underwater')

vid1.compile_vid(fileName='topclips.mp4')
```

We first import the AVE class and declare our first video editing thread, vid1. Although multiple video editing threads can be created and compiled in a single file, AVE is a memory-intensive application and the application may crash. As a rule of thumb, the more videos that are loaded, the more memory gets consumed and as more queries are added, the more VRAM is consumed.

There are two functions that are used to create the video: *addVideo()* and *newQuery()*. *addVideo()* takes a string with a path to a video file (.mp4 videos are required). The order in which *addVideo()* is called does not matter. *newQuery()* takes a string written by the user of a clip that they want to extract from the videos they have loaded. The order in which *newQuery()* calls are made is important. In the example above, the final video will first show “petting a cat,” then “a small white penguin,” and lastly “a penguin underwater.”

Finally, the function *compile_vid()* accepts a string that will be the final file name and takes all the searched clips and aggregates them into a single file that is exported. Video processing only begins once *compile_vid()* is called. The functions *addVideo()* and *newQuery()* only collect data that will be processed once the compilation process has begun.

4. Performance

This section will discuss AutoVideoEdit’s performance, strengths, and weaknesses. The specifications of the test system are as follows: 16 GB RAM, RTX 3050TI 4 GB VRAM, and Windows 11.

4.1 Effect of Video Length

During this test, we randomly sampled YouTube videos of various lengths between zero and 10 minutes translating to 1,000 to 14,000 frames. A graph showing the performance of video preprocessing (blue) and query processing (red) can be seen below in Figure 2:

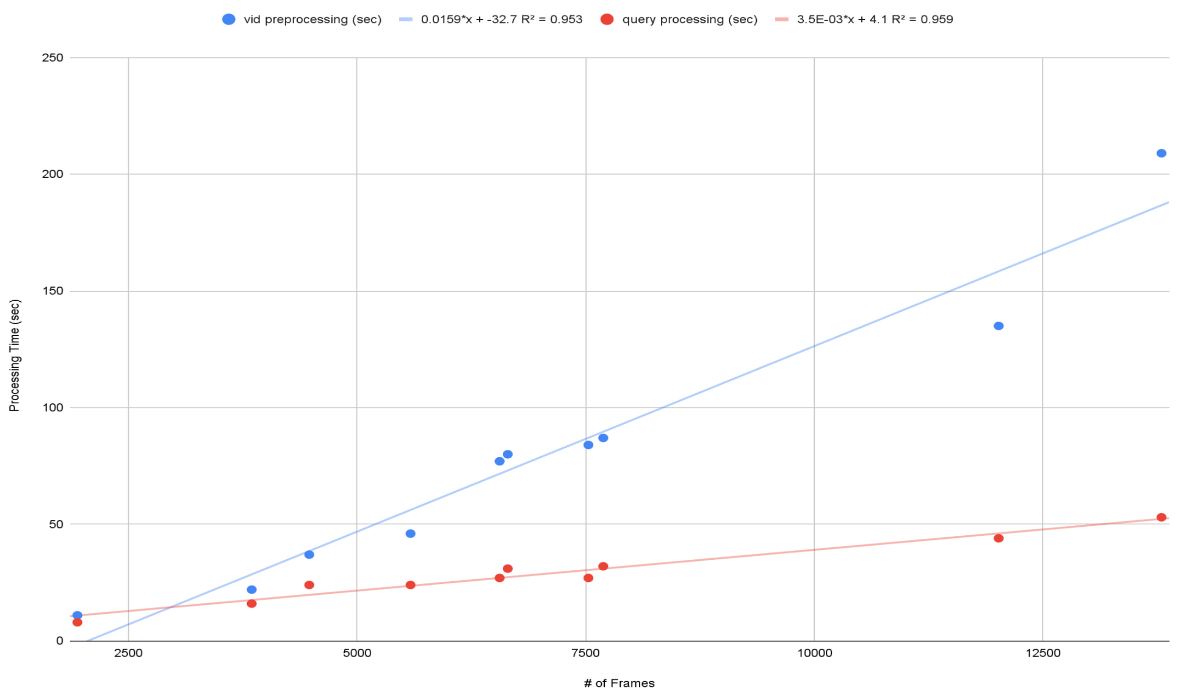

Figure 2. Processing time vs frame count. Processing time across frame count for video preprocessing (blue) and query processing (red).

During this test, we watched each video and provided a single general visual search query relevant to the video. During each test, a single video preprocess cycle and a single query process cycle are made (recall that the total number of run-throughs will be equal to the product of the number of videos and the number of queries). All the data points shown in Figure 2 were successful processes in which the output of the algorithm was manually validated with the search query. It should be noted that we also tested AVE on longer videos, however, we were limited by our test system's memory. By limiting the algorithm to under 15,000 input frames, we can establish a roughly linear time complexity.

Another observation is that the video preprocessing stage takes significantly longer than the query processing stage. This is a consequence of utilizing PySceneDetect. During the query processing cycles, only CLIP–which is hardware accelerated by the use of a dedicated GPU–is used. However, video preprocessing cycles use PySceneDetect which instead relies heavily on system memory to run. This issue is probably why our test system was unable to process videos longer than 15,000 frames.

4.2. Testing Video Search Variety on Query

One of the largest considerations while using AutoVideoEdit is its accuracy when picking clips, especially over a variety of videos. To visualize this, we can plot the cosine similarity score between a query and a frame for every frame. For example, with the input video: https://youtu.be/YvT_gqs5ETk and query "lady looking out of a window," we are presented with the graph shown in Figure 3.

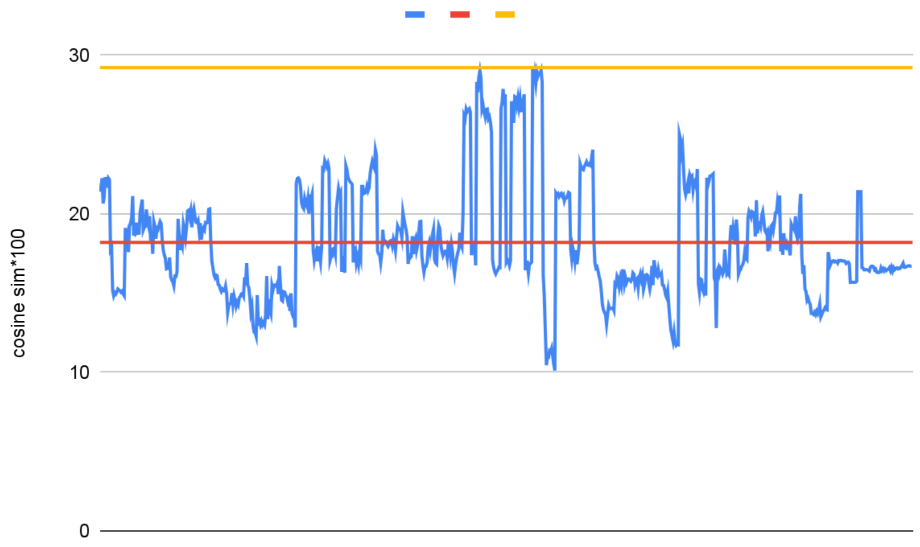

Figure 3. Cosine similarity graph (cos_sim*100) between a query and frame for every frame. Average similarity score of 18.2 (red) and a maximum score of 29.2 (yellow).

Figure 3 indicates that multiple instances of the video match the target query as shown by the multiple peaks that reach the maximum similarity value (i.e., high accuracy). In situations where the algorithm cannot find a frame that matches the query well, the graph will look more similar to Figure 4.

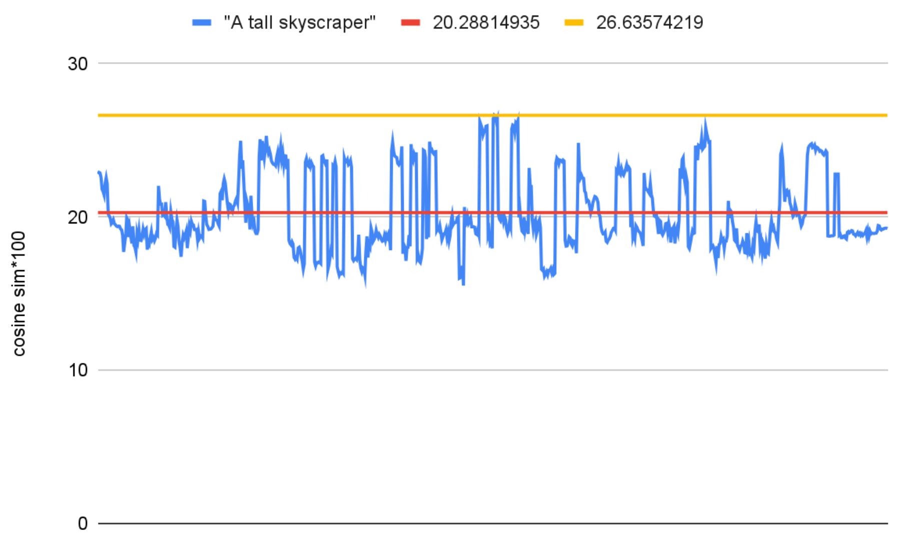

Figure 4. Cosine similarity graph (cos_sim*100) between a query and frame for every frame

In Figure 4, we can see that a greater number of smaller peaks reach or approach the maximum value due to the greater uncertainty that the algorithm has to handle along with far lower accuracy. We can aggregate multiple queries into one graph to display how each search is represented in a single video, as shown in Figure 5.

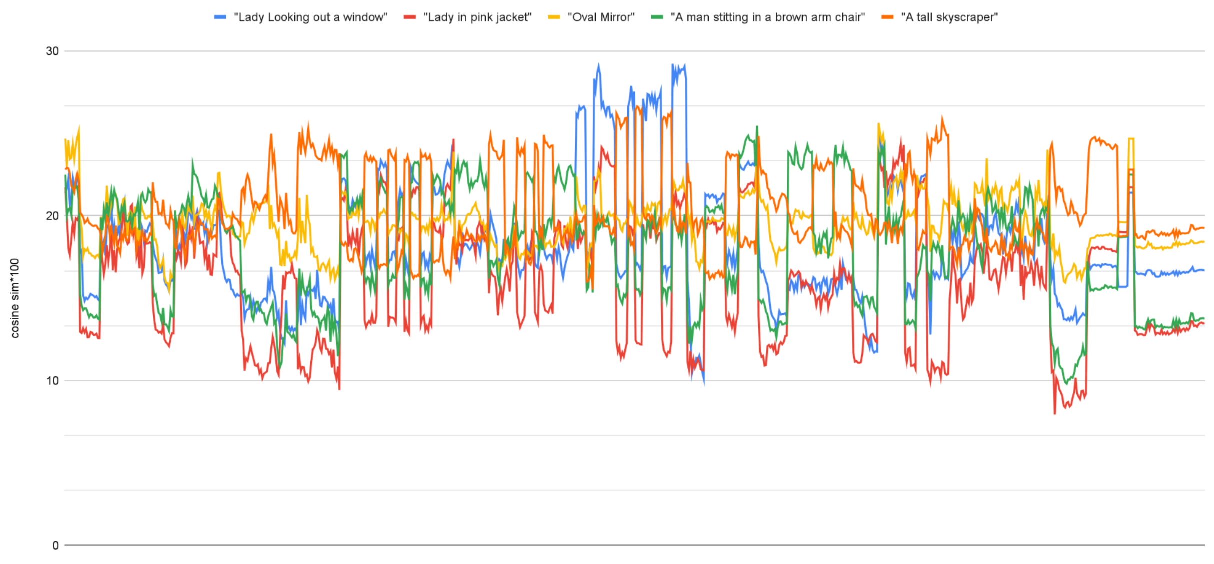

Figure 5. Comparison of cosine similarity (cos_sim*100) across five queries for the same video. Queries distinguished by color.

4.3. Review of Strengths and Limitations

AutoVideoEdit excels in situations where there is a specific scene that is easily identifiable and definable. That is, descriptions that only appear once are easier to locate since it removes the possibility of receiving a clip that may also have a strong similarity to the query but is not the intended clip.

An example of this is searching through the video: https://youtu.be/rzjGz20WUyM with the query “Red cups on a table.” The algorithm is able to correctly search for the clip containing the frame shown in Figure 6.

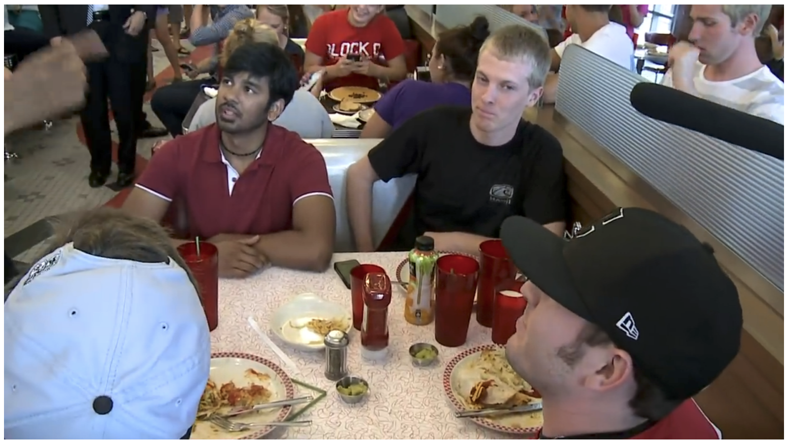
Figure 6. An example of a correct output from AVE.

This is a specific description that only appears once within the entire video. Additionally, this example also demonstrates the characteristics of identifiable and definable. The cups are clear in image quality and are easy to define with a consistent visual form. An example of unidentifiable and undefinable inputs is the video: https://youtu.be/hW3KpAY30p0 with the query: “robot holding pack of cards.” Figure 7 shows the predicted vs observed outputs (single frames shown from entire clips).

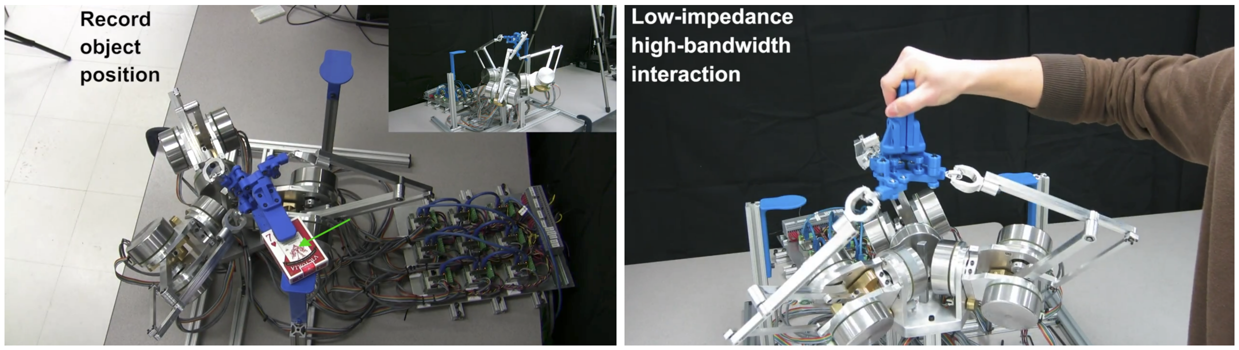

Figure 7. Side-by-side comparison of predicted and observed outputs. Left, predicted output. Right, observed output

We predicted the algorithm to perform correctly by showing a clip where the blue grippers are grasping the pack of cards. However, the observed output of the algorithm is of researchers directly manipulating the robot. In this situation, robots can have many forms making them difficult to define when writing a query, additionally, the “pack of cards” is difficult to identify and separate from other generic objects.

Videos that perform well are those that have many discrete common objects. For example, cooking videos perform very well since they have many discrete identifiable objects that can be inserted into a query such as “cooking a steak” or “cutting lettuce.” On the other hand, videos that are repetitive, difficult to identify, or lack generality do not perform well. Examples of these types of videos include robotics (as seen above), sports of most kinds, and static footage where there are very few changes between each frame (e.g., a video of someone sitting down and talking to a camera).

5. Review

5.1. Future Work

As discussed in the motivations sections, AutoVideoEdit aims to be a tool that streamlines the majority of time-consuming tasks related to video editing. The tool currently tackles the issue of clip searching; however, there is still more room for improvement. First and foremost, performance can be significantly improved by making better use of a dedicated GPU during the video preprocessing stage. This would also allow us to process longer videos on devices with lower specifications. Further, this application could also benefit from multithreading, especially while processing queries. Since the total number of query processing cycles is equal to the product of the number of videos and the number of queries, this process would benefit from being performed in batches rather than individually.

In terms of increased functionality, there are other methods of clip searching that can be implemented, such as searching by audio (matching text or description), searching by optical character recognition (OCR), or searching by proper nouns or tagged items. These features would allow the user to have more control over the end product while only having to input their ideas through natural language and allow AutoVideoEdit to overcome some of its current limitations.

Finally, there are a plethora of editing tasks that could benefit from automation. For example, syncing audio and video. In a situation where a user wants to sync their data from an external microphone to a video of someone speaking, or where a user wants to move their video along with the beat of a song, both of these tasks have the potential to be automated. This is the direction that AutoVideoEdit intends to strive towards.

5.2. Conclusion

In this report, we present the AutoVideoEdit Python package which provides user-friendly automation for tasks related to video editing. We demonstrate that AutoVideoEdit can successfully identify and merge clips from various types of videos that match a user-provided query. In particular, AutoVideoEdit performs best on videos with an easily identifiable scene but struggles on videos without discrete objects or are repetitive.

By processing randomly sampled video clips with varying numbers of frames and plotting the processing time by the number of frames in each video clip, we found that our algorithm has a linear time complexity when limited to under 15,000 frames during video preprocessing.

Our future work is dedicated to optimizing GPU usage and implementing multithreading which will improve our current limitations with regard to performance. Additionally, more work can be done in implementing a greater number of search methods to assist current search limitations.

Bibliography

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, & Ilya Sutskever. (2021). Learning Transferable Visual Models From Natural Language Supervision. *ArXiv: Computer Vision and Pattern Recognition*. https://doi.org/10.48550/arXiv.2103.00020

Brandon Castellano. (2014-2022). PySceneDetect Documentation. Scene Detection Algorithms. https://pyscenedetect.readthedocs.io/en/latest/reference/detection-methods