Cross Cultural Analysis System Improvements

Guandong Liu
Columbia University
gl2675@columbia.edu

Yiyang Zeng
Columbia University
yz3645@columbia.edu

1 Introduction

Cross-cultural analysis investigates the new task of determining which textual tags are preferred by different affinity groups for news and related videos. We use this knowledge to assign new group-specific tags to other videos of the same event that have arisen elsewhere. We map visual and multilingual textual features into a joint latent space using reliable visual cues, and determine tag relationships through deep canonical correlation analysis (DCCA). We are interested in any international events such as epidemics and transportation disasters and detect country-specific tags from different countries' news coverage [1].

Our current pipeline is able to collect data, preprocess data, and analyze data. This semester, we investigated and improved system performances in different aspects to make it more reliable and flexible. This report consists of two parts: the first part gives a thorough overview to the whole pipeline and the second part talks about several bug fixes and performance improvements of the current system.

2 Pipeline User Manual

This section gives a thorough overview to the whole pipeline and goes through each step with AlphaGo in Chinese and English task as an example.

2.1 Get Video Metadata

The pipeline starts from getting video metadata from various sources. In our experiments, we got video metadata about AlphaGo from Chinese sources (Tencent, Iqiyi, Youku, BiliBili) and English source (YouTube). Note that to get video metadata from YouTube, one must first enable YouTube Data API and get own API credentials (`key.txt` is the API credentials).

```
python3 $src/spidering/get_metadata.py "AlphaGo" -n 5
-o $results/AlphaGo/chinese/video_metadata -s qq iqiyi youku bilibili
python3 $src/spidering/get_metadata.py "AlphaGo" -n 5
-o $results/AlphaGo/english/video_metadata -s youtube -k key.txt
```

2.2 Download Videos

After getting video metadata, the pipeline produces a topic (AlphaGo) directory and each language has a sub-directory which contains several metadata files. The pipeline downloads videos based on metadata.

```
python3 $src/spidering/download.py $results/AlphaGo/chinese/video_metadata/metadata
-ao $results/AlphaGo/chinese/audios -vo $results/AlphaGo/chinese/videos -s bilibili qq iqiyi
python3 $src/spidering/download.py $results/AlphaGo/english/video_metadata/metadata
-ao $results/AlphaGo/english/audios -vo $results/AlphaGo/english/videos -s youtube
```

2.3 Extract Audio

It is then followed by several extraction steps. First, it uses `ffmpeg` to extract audio from videos.

```
python3 $src/dataset/extract_audio.py $results/AlphaGo/chinese/videos
```

```
-o $results/AlphaGo/chinese/audios
python3 $src/dataset/extract_audio.py $results/AlphaGo/english/videos
-o $results/AlphaGo/english/audios
```

2.4 Extract Transcript

Transcript extraction requires Google Cloud Storage JSON API [2] and Cloud Speech-to-Text API [3]. To extract transcript, one first needs to create a storage bucket on Google cloud platform and Google cloud API credentials (different from the one in getting video metadata step). Users can optionally provide hint words for speech recognition.

```
python3 $src/dataset/extract_transcript.py $results/AlphaGo/chinese/audios key.json
[gcp-bucket-name] -o $results/AlphaGo/chinese/transcripts
-l cmn-Hans-CN -d audios_cn -t "AlphaGo" "AlphaZero" "DeepMind"
python3 $src/dataset/extract_transcript.py $results/AlphaGo/english/audios key.json
[gcp-bucket-name] -o $results/AlphaGo/english/transcripts
-l en-US -d audios_en -t "AlphaGo" "Lee Sedol" "Ke Jie"
```

2.5 Process Transcript

The pipeline preprocesses transcripts to prepare for extracting frame. It produces a JSON file which includes (timestamp, token) pairs for every video.

```
python3 $src/dataset/process_transcript.py $results/AlphaGo/chinese/transcripts
-o $results/AlphaGo/chinese/processed_transcript.json
python3 $src/dataset/process_transcript.py $results/AlphaGo/english/transcripts
-o $results/AlphaGo/english/processed_transcript.json
```

2.6 Extract Frame

It then extracts frames from processed transcripts and videos and creates a sub-directory for each video.

```
python3 $src/dataset/extract_frame.py $results/AlphaGo/chinese/processed_transcript.json
$results/AlphaGo/chinese/videos -o $results/AlphaGo/chinese/frames
-d $results/AlphaGo/chinese/image_text_pairs
python3 $src/dataset/extract_frame.py $results/AlphaGo/english/processed_transcript.json
$results/AlphaGo/english/videos -o $results/AlphaGo/english/frames
-d $results/AlphaGo/english/image_text_pairs
```

2.7 Extract Feature

The pipeline uses the VGG-19 network as feature extractor. It takes the last linear layer and processes frames in batch. Note that it resizes the frame resolution to adapt the VGG-19 network. After iterating over the input folder, it produces a feature file for each video.

```
python3 $src/dataset/extract_feature.py $results/AlphaGo/chinese/image_text_pairs
-o $results/AlphaGo/chinese/features_data
python3 $src/dataset/extract_feature.py $results/AlphaGo/english/image_text_pairs
-o $results/AlphaGo/english/features_data
```

2.8 Find Duplicate

After getting feature data for each video in both languages, the pipeline finds nearly duplicate frames from extracted features. By default, it uses angular distance for similarity but it can also be L1 or L2. Users can also provide a threshold of similarity.

```
python3 $src/dataset/find_duplicate.py $results/AlphaGo/chinese/features_data
$results/AlphaGo/english/features_data -o $results/AlphaGo/pairs.json -t 0.25 -d angular
```

2.9 Deep Canonical Correlation Analysis

The last step, Cross cultural analysis, consists of an embedding model and a relevance model. We use existing language/vision models (e.g. ResNet/Word2Vec/BERT, as in the /embedding folder) to map the raw pre-processed data (image or text) to the embedding vector. Then, DCCA is our default choice of relevance model which learns the correlation between the two embedded vectors [4].

3 Improvements

The improvements on the pipeline focus on three aspects:

- **Tool Building**: A tool that can monitor the time/space usage of the pipeline was built so we can better identify bottleneck in the pipeline.
- **Bug Fixing**: There were a couple of bugs in the code repository that prevented the pipeline from operating normally. These bugs are either caused by outdated formats of APIs or inconsistent versions but are fixed now.
- **Performance Optimization**: The ways in which some pipeline components were implemented were highly computation/memory consuming. Those components were optimized so they can now run on not necessarily insane hardware but mediocre hardware.

3.1 Time and Space Analysis Tool

3.1.1 Functionality

A tool which can monitor the time (in seconds) and memory usage (in GBs) of a pipeline component (usually a function) was built. The tool takes a function and the input for the function as the input of the tool, runs the function in a new thread and finally measures the time and memory usage of the function. At the end, the tool will display the measurements in the console in a human readable format. In such way, the time and memory usage of pipeline components can be easily monitored and bottlenecks in the pipeline can be clearly identified.

3.1.2 Potential Improvement

Currently, the tool relies on measuring the memory usage of the main process of the pipeline, before and during the execution of measured functions. Then, the memory usage before the execution is subtracted from the maximum memory usage during the execution to get the maximum memory usage of measured functions.

This method works, but with some problems:

- Due to the inefficient garbage collection of Python, the "starting memory usage" might not be accurate, since the memory used by the previously measured functions might not be properly released yet, resulting in a lower estimate of memory usage of the currently measured function.
- The tool measures the memory usage of the entire process rather than the specific thread on which the measured function runs. This is potentially a problem because if, while the measured function is being executed, the main process somehow starts other threads, which also take some memory space, the final measured memory usage might be inaccurate.

There might be a solution to these problems, but during time investigating the problems, we did not figure it out.

3.2 Video Metadata

Video metadata consists of some important video properties such as URL and title. This semester, we improved the reliability of getting video metadata and fixed bugs for getting metadata from various Chinese video sources. Previous code assumed getting video metadata could be done successfully. But we did some experiments and found it could occur errors and stop the whole pipeline. To address this problem, we added a `try-except` block to capture any error during fetching video metadata process and report to users.

3.2.1 YouTube

When we get video metadata from YouTube, we need to call Google API with specific parameters. There is a parameter called **relevanceLanguage** which instructs the API to return search results that are most relevant to the specified language. The parameter value is typically an ISO 639-1 two-letter language code. Besides, results in other languages will still be returned if they are highly relevant to the search query term [5]. Since we only use YouTube to search videos in English, we set this parameter to `en-US` so that search results are highly related to English. Without this parameter, search results may be mixed with different languages. To use YouTube to search videos in other languages in future, pass **relevanceLanguage** as a method parameter.

3.2.2 Tencent, Iqiyi, and Youku

The previous implementation of video metadata fetching was outdated, so some video sources stopped functioning. The impacted sources were: Tencent(QQ), Iqiyi and Youku.

- **Tencent and Iqiyi**: The metadata matching algorithm (the algorithm to find video metadata in web pages) was outdated. We updated the algorithm and the problem is fixed.
- **Youku**: Youke has Captcha which blocks API requests from the pipeline, which we were unable to fix.

Note that the metadata matching algorithm might need updates fairly frequently.

3.3 Video Downloading

The work on video downloading focused on improving the robustness of the process. Before optimization, the video downloading process only tried to download each video once. If the first attempt failed, the video was skipped. In such way, we saw a lot of random failures of video downloading attempts. We optimized the process by giving a video multiple chances, with some time gap between two attempts. After the optimization, we discovered that virtually all random failures were eliminated, with download failures limited to only inaccessible videos.

3.4 Audio Extraction

`ffmpeg` is the tool we use to extract audio from videos. The previous command used to extract audio was:

```
ffmpeg -y -i [video_path] -f [audio_type] -r [sampling_rate] -ac [audio_channel] [audio_path]
```

From `ffmpeg` official website, options are applied to the next specified file [6]. `-r` option is applied to the audio file but there is on such option in audio options and the output audio file without this option is identical to output with this option. Besides, the previous command was also time-consuming. Thus, we discarded `-r` option and changed to the following command:

```
ffmpeg -y -i [video_path] -f [audio_type] -ac [audio_channel] -vn [audio_path]
```

Where `-vn` is used to skip inclusion of video and speed up audio extraction. If we have to manually set audio sampling rate in future, please use `-ar` option to set the audio sampling frequency.

3.5 Transcript Extraction

According to the latest Python docs, **timedelta** objects take `microseconds` argument instead of `nano` argument now and thus we updated the corresponding codes.

3.6 Frame Extraction

The process of frame extraction was in dire need for optimization because the original process used way too much memory and hard drive space.

The original frame extraction process went through the following steps:

1. Go over a video and extract frames (saving to hard drive) according to the transcript
2. Construct metadata that contains metadata and frame data.

In our optimization, step 1 kept unchanged. After all, all frames have to be saved to hard drive. However, step 2 is way too wasteful, because it saves frame data again, given that the data is already saved in step 1. To save hard drive space, we decided not to store frame data in step 2. Instead, we wrote the path to frame files into the metadata, so when we need to get frame data according to the metadata, we could read from the path stored in the metadata.

Also, the original method processed all videos in one run and stored the metadata in one large binary file. The problem with this method was that the metadata file would take up too much memory space so the process would crash (with a 4 minute video on a 16GB memory machine). We changed this to one metadata file per video and frame extraction produced a folder of metadata files now.

3.7 Feature Extraction

Original feature extraction process generated one large binary feature file and could also cause out-of-memory (OOM) errors. As discussed in section 3.6, we implemented a new version of frame extraction. To adapt the new version of frame extraction, we took the metadata directory (output folder of frame extraction) as input and saved feature data to a separate file for each video. We also resized each frame first to save memory and processed frames in batch to improve efficiency.

3.8 Deep Canonical Correlation Analysis

Originally, Deep Canonical Correlation Analysis (DCCA) was absent from the pipeline. Only part of the code was in a separate folder in the project repository. We adapted the code to work with our pipeline integration. After that, we did some initial testing with the DCCA code with our test input. However, due to lack of cloud credits, we were unable to test the code extensively so that part could be the first step in the next semester.

Another thing could do is DCCA visualization. To interpret the result of DCCA, we could decompose the correlation matrix to get the visual vector and the language vector and investigate how they relate each other.

4 Conclusion

During this research, we thoroughly inspected the source code of the pipeline, identified several improvements we could make and implemented them. After our optimization, the pipeline could progress much faster and more stably. Additionally, the pipeline can be run on a computer with a much lower configuration.

In the mean time, we also fixed many buggy/outdated code that could severely interrupt the pipeline.

Finally, we integrated the pipeline components into one main file. Through the main file, the pipeline can be run and finished in one command, with the benefits of being capable of being attached with a debugger that can significantly simplify the developing process of the pipeline.

On the flip side, we also had some difficulties doing the research. To effectively run the pipeline, we needed a computer with a decent discrete graphics card. One simple way to obtain such machine is using Google Cloud credits. However, we only got our hands on such credits when the semester was almost over, so the progress on some tasks in improving the pipeline was stagnant. In the future, to effectively carry out the tasks, such as integrating DCCA, getting the GCP credits should be top priority.

5 Resources

- GitHub repository: `https://github.com/zyyhhxx/Cross-Cultural-Analysis2`
- AlphaGo dataset: The AlphaGo dataset is available at `https://drive.google.com/drive/folders/1jm-wcG4i_2OlrhBd_1JxCfq65CRLDCJY?usp=sharing`. In the drive, you will find both English and Chinese video clips in the /english and /chinese folder, and there are several more video sources in the /raw folder. They should be around 20-30G per topic, and should be around 100G after you preprocess it.

References

[1] Tagging and browsing videos according to the preferences of differing affinity groups.

[2] Cloud storage json api.

[3] Cloud speech-to-text api.

[4] Stanley Liu. Tagging and browsing videos according to the preferences of differing affinity groups research report.

[5] Youtube data api.

[6] ffmpeg.