Paper Title

Author Name, Author Name

Organization

Abstract

The rapid advancement of Generative AI (GenAI) has revolutionized the content creation process across text, images, videos, and music, making it faster and more efficient than ever before. However, this growth also brings challenges, particularly in managing and identifying duplicated or similar content. In this project, we address a key challenge in video matching through the implementation of a video copy detection system. Our primary objective is to deploy an Information Retrieval Engine capable of identifying relevant and similar videos from a given database. For example, consider a short input clip, such as a news segment registered by a journalist. Our system will retrieve similar or related videos from a broadcasting company's database, enabling the creation of official news videos enriched by existing content.

Contents

1 Introduction

Key Tasks: This report includes 5 sections: dataset, preprocessing, models, infer and our result on test datasets.
This project has used datasets from MetaAI for training as well as validation (with ratio 8:2) and integrates advanced deep learning techniques to achieve video copy detection. As the result, we tested the model on some datasets (one from MetaAI, one is created by us with edit tool) and finally got some good score on these datasets.

---

2 Dataset

The datasets we are using

2.1 Data Structure

The dataset is structured as follows:

- **feat_zip**: Contains `feats.zip`, generated during training video scores.
- **jpg_zips**: Includes images extracted from videos at 1 frame per second (fps). This will be generated in processing phase.
- **lmdb**: Stores images from `jpg_zips` and is organized as follows:
  - `dics`
  - `train_vsc`
  - `vsc`
- **meta**: Contains metadata files in Excel or text format:
  - **Train**:
    - `train_vids.txt`: IDs of all training videos.
    - `Train_query_ids.txt`: Query video IDs for training.
    - `train_ref_vids.txt`: Reference video IDs for training.
    - `train_query_metadata.csv`, `train_reference_metadata.csv`: Video features such as duration (sec), frames per second (fps), and resolution (width, height).
    - `train_ground_truth.csv`: Query-reference video pairs with specific timestamps.
    - `train_positive_query.txt`: IDs of copied query videos.
  - **Validation**:
    - `Val_query_ids.txt`: Query video IDs for validation.
    - `Val_query_metadata.csv`: Metadata for validation queries.
    - `Eval_matches.csv`: Query-reference video pairs with specific timestamps.

  - **Test**:
    * `test_query_ids.txt`, `test_ref_vids.txt`: Query and reference video IDs for testing.
    * `test_query_metadata.csv`, `test_reference_metadata.csv`: Metadata for test queries and references.
    * `test_ground_truth.csv`: Query-reference video pairs with specific timestamps.
  - `vids.txt`: General list of video IDs.
- **images**: Contains background images, including emojis (`bg_img`).
- **videos**: Video files split into `train` and `test` subsets:
  - **Train**: Includes `query` and `reference` videos.
  - **Test**: Includes `query` and `reference` videos.

***Notice that** for three stages of train, val, test, meta has provided 3 unique datasets including query and reference video. However, in the project, we have split the original train dataset with the ratio 8:2 into train_query_ids.txt and val_query_ids.txt. Therefore, the ID of val ref videos are defined in meta/train/train_ref_vids.txt.*

2.2 Folder Structure

```
data/
|-- feat_zip/
|   |-- feats.zip
|-- jpg_zips/
|-- lmdb/
|   |-- dics/
|   |-- train_vsc/
|   |-- vsc/
|-- meta/
|   |-- train/
|   |-- val/
|   |-- test/
|-- images/
|   |-- bg_img/
|-- videos/
    |-- train/
    |-- test/
```

Figure 1: Dataset Folder Structure

3 Preprocessing

Preprocess data before building model

3.1 Video to images

Because our model is based on image data, we convert the original video to jpgs, at fps = 1. The converted videos are put into zip files numbered 00-99, corresponding to the last 2 digits of their IDs. The steps are as follows:

- Input metadata csv for all videos
- Input list of IDs for videos that will be converted
- Search metadata csv for videos that match the input ID list.
- Convert videos into jpegs and put them into corresponding zip files using ffmpeg.

3.2 Images to lmdb

Lightning Memory-Mapped Database (LMDB) is an embedded transactional database in the form of a key-value store. We use LMDB instead of other type of search engine like **Elastic Search** because:

- **High Read Performance:** LMDB uses memory-mapped files, enabling fast data access, suitable for video copy detection tasks.
- **Simplicity:** LMDB is an embedded database, easy to deploy, allowing focus on processing algorithms.
- **Memory Optimization:** LMDB consumes minimal resources, ideal for handling large video data.
- **No Text Searching Needed:** LMDB can store binary data (like images) directly as values in the database.
- **ACID Transactions:** LMDB ensures data integrity, suitable for tasks requiring high precision.

The steps are as follows:

- Choose size of lmdb file
- Input list of IDs
- Put videos into lmdb

Notice that *Some files use the jpeg zips directly, while some take data from the LMDB file. For further clarification, consult other sections..*

---

4 Train_vid_score

In this folder, we address two tasks.

4.1 Embedding Extraction

First, we need to extract all the frames (images) into embeddings. The input consists of two folders located in `../data`: `jpg_zips` and `vids_txt`.

- `jpg_zips`: Contains images of videos (created during the preprocessing phase).
- `vids_txt`: Contains the names of the videos (training data only).

The models used for this task are:

- `clip_vit_l_14` (from GitHub)
- `clip-vit-base-patch32` (from Hugging Face)

The rationale for using two models is to avoid relying on their pre-trained models and instead generate our own embeddings. The outputs are stored in `../data/feats.zip`, which contains the video names and their embeddings.

This step was successfully implemented using two scripts:

- `my_extract_feat_git`
- `my_extract_feat_hf`

Finally, we transform model CLIP to .pt format for inference section.

4.2 Train Video Score

The next task involves using the `train_vid_score` file to determine whether a video is edited or not.

Inputs

The input for this step is the embeddings generated in the previous phase.

Outputs

The output is a log file containing metrics such as loss, Average Precision (AP), and accuracy.

Models Used

The models employed for this step are:

- `Chinese-RoBERTa` (from GitHub)
- `google-bert` (from Hugging Face)

Process

1. Transform `feat_dim` (embedding dimensions from the CLIP model) to `bert_dim` for compatibility with the RoBERTa model.
2. Add tokens `[CLS]` and `[SEP]`:
    - `[CLS]`: Summarizes the content of the video.

- `[SEP]`: Marks the end of the sequence.

3. Create a mask for valid frames to exclude noisy frame calculations.
4. Feed embeddings into the `Chinese-RoBERTa` model to learn the video sequence scenario and extract video information.
5. Generate two types of pooling:
   - `avg_pool`: Average summary of valid frames.
   - `cls_pool`: Hidden state of `[CLS]` summarizing the main video content.
6. Combine the pools into `cat_pool`, a vector containing both summarized and detailed video information.
7. Pass `cat_pool` through a linear layer (`nn.Sequential(cat_pool, 1)`) to generate logits.
8. Classify videos:
   - If `logit > threshold`, classify as an edited video.
   - Otherwise, represent as a random vector with small values.
9. Compute binary cross-entropy loss between logits (sigmoid) and labels (`label = 1` for copied, `label = 0` for uncopied).
10. Calculate additional metrics such as accuracy, precision (`pn`), and Average Precision (AP) on the training and validation sets.
11. Transform model Chinese_roberta to .pt format for inference section. (vsm.torchscript.pt)

Experiments

Three configurations were tested:

- **Git + Git:** CLIP (GitHub) + RoBERTa (GitHub)
- **HF + HF:** CLIP (Hugging Face) + BERT (Hugging Face)
- **Git + HF:** CLIP (GitHub) + BERT (Hugging Face)

Observations

- Configurations 1 and 3 were successful (loss, accuracy, and AP fluctuated).
- Configuration 2 resulted in constant validation accuracy, indicating an issue with the Hugging Face CLIP model.
- Notably, Hugging Face CLIP embeddings have dimensions of 512, while GitHub CLIP embeddings have dimensions of 1024.

---

5 Train_v115

Train Embedding

5.1 Build dataset

Unlike train_vid_score, train_v115 has a different way to build a dataset. The detail will be describe below:

- **Transform_q: hard_pipeline** - Applies strong augmentations such as flip, rotate, crop, noise, blur, scale, perspective, text overlay, emoji overlay, color jitter, etc.
- **Transform_k: normal_pipeline** - Applies lighter augmentations such as flip, crop, color jitter, rotate, noise, blur, and scale.
- **Transform_n: native_pipeline** - Applies simpler augmentations such as flip, crop, noise, and image compression.

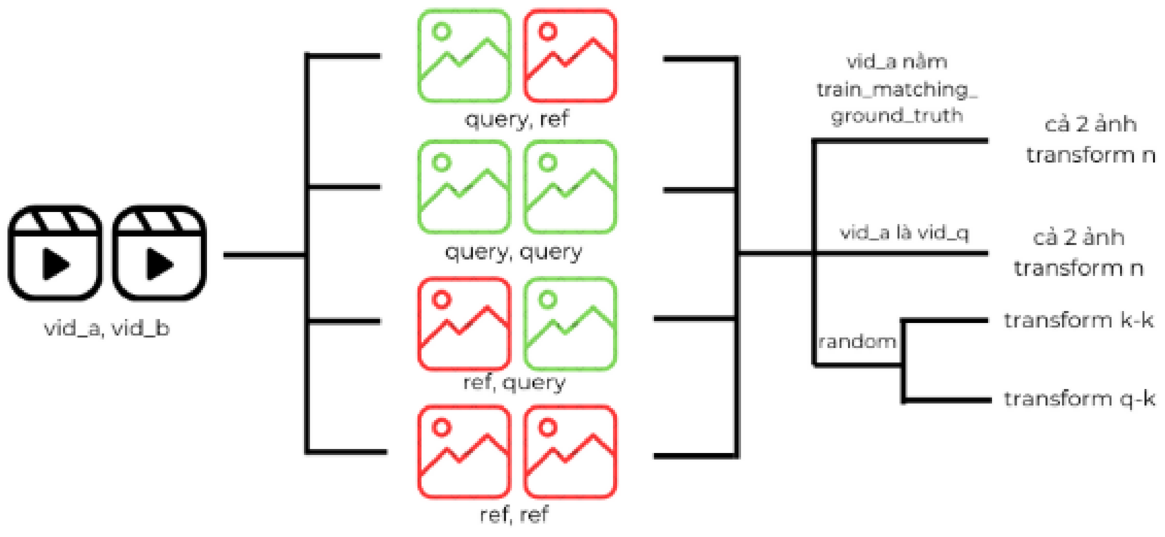

Figure 2: Build dataset pipeline

Initially, 2 videos selected by index in the DataLoader function. Extracting an image from each video. There will be 4 cases when randomly taking 2 videos (**Figure 2**). From these 4 cases, we will have 3 cases of image processing as follows. In the first case, if vid_a is in the ground truth, then both images will be transform n. Or if vid_a is a video query, then both images will also be transform n. The remaining cases will randomly choose between both images being transform k or 1 image being transform q, 1 image being transform k.

5.2 Model Swin_v2

After loading the data, we feed it into Swin_v2 to learn how to embedding datas. Our Swin_v2 is trained with two loss functions: contrastive loss and entropy loss. **Contrastive loss** encourages similar embedding vectors (e.g., embeddings of similar or related data points) to be closer together in the vector space. Meanwhile, dissimilar embeddings (embeddings of unrelated data points) are pushed away. **Entropy loss** is calculated based on the similarity matrix between embeddings, which helps to avoid similar but not exact pair embeddings from being too close together. This can make embeddings more distinguishable, especially in unsupervised deep learning models.

The Contrastive Loss. We use the Contrastive loss, which is softmax cross-entropy loss with temperature. The loss function is formulated as follows:

$$L_{\text{InfoNCE}} = -\frac{1}{|P|} \sum_{i,j \in P} \log \frac{\exp\left(\cos(z_i, z_j)/\tau\right)}{\sum_{k \neq i} \exp\left(\cos(z_i, z_k)/\tau\right)}$$

Where P is the set of positive pairs, z_i represents the descriptor, τ is the temperature. (1)

The Entropy Loss. We using the entropy loss. The loss function is formulated as:

$$L_{\text{KoLeo}} = -\frac{1}{N} \sum_{i=1}^{N} \log \left(\min_{j \neq i} \|z_i - z_j\| \right)$$

Where N is the size of the training set. (2)

The Final Loss. The final loss function is:

$$L = L_{\text{InfoNCE}} + \lambda L_{\text{KoLeo}} \tag{3}$$

Where λ is the weights of Entropy Loss term.

After train 40 epoch of Swin_v2 training, we have weight in .pth format. We transform it in to .pt format for inference section.

---

6 Inference

Small description

6.1 Query

Input and Output:

- **Input:**
  - Checkpoint files in TorchScript format for models (CLIP, VSM, SwinV2).
  - Data folder in `.mp4` format and `metadata.csv` file.
  - `.npz` file for normalization (use `train_refs.npz` here).
- **Output:** `test_query_sn.npz` file containing extracted features of the test query videos.

Pipeline:

- **Dataset:**
  - **VideoDataset:** Each entity represents a video containing: `video_id`, `timestamp`, frames, and results of `transform1` and `transform2`.
  - **Transform1:** Performs initial transformations such as resize, `to_tensor`, and basic normalization for the input of the `video_score_model`.
  - **Transform2:** Similar transformations tailored for the SwinV2 model.

- **Function Process:**
  - **Input:** One video.
  - Use the CLIP model to extract features (max 256 frames). If the number of frames is smaller, apply padding. These features are only used in the `video_score_model`.
  - **Output of CLIP model:**
  - Use the SwinV2 model to extract features, which will be used in subsequent steps.
  - Apply the `video_score_model` to score each video:
    - If the score exceeds a very small threshold (0.001): Normalize using L2 norm, calculate the average correlation of one frame with others in the same video. Remove frames where the correlation with other frames exceeds 0.975 to reduce redundancy and computational overhead.
    - If the score is below the threshold, create a random feature for the video at time 0 seconds.
- **Normalization:**
  - Remove the dimension with the lowest variance (among the 512 dimensions) — optional step.
  - Perform L2 normalization for both the query and the `norm_file`.
  - Use `CandidateGeneration` and `MaxScoreAggregation` to build the index.
  - For each query: Use KNN to find the closest match in `score_norm_refs`.
    - If the score obtained from `video_scores` is below the threshold, multiply it by -100 (reduce the influence of normalization).
    - If the score satisfies the condition, compute `norm_term` = similarity between the query and the closest match in `score_norm_refs`.
  - Finally, add the normalized feature with `norm_term` to produce the final result. The output includes the "closeness" or "context" of the query video.

Notice that *Index - Cluster* `score_norm_refs` *to perform faster KNN-based searches.*

6.2 Ref

Input and Output:

- **Input:**
  - Data folder in `.zip` format (images) and metadata file.
  - SwinV2 checkpoint file in TorchScript format.
  - `.npz` file for normalization (use `train_refs.npz` here).
- **Output:** `test_ref_sn.npz` file containing extracted features of the reference test videos.

Pipeline:

- **Dataset:**
  - `D_vsc`: Each item includes `video_id` and `frame_tensor` (S, C, H, W), where S is the number of frames.
  - Images are read directly from `.zip` files without significant preprocessing, ensuring faster execution compared to the query.
  - A `collate_fn` function is provided to batch the data.
- **Feature Extraction:** Use the SwinV2 model to extract features.
- **Normalization:** Normalize using the `normalize` function from `sklearn.preprocessing`.

6.3 Evaluation Metrics

Input and Output:

- **Input:**
  - `test_ref_sn.npz` and `test_query_sn.npz`.
  - Ground truth file.
- **Output:** Candidate file listing scores for video pairs in descending order and the overall uAP score.

Pipeline:

- **Matching Function:**
  - Use the `search` function to return candidates.
  - Two key parameters depend on the number of videos in the test set:
    - `Retrieve_per_query`: Average number of reference videos retrieved per query video. Multiplied by the number of query videos to determine the `global_k` parameter (total reference videos for computation).
    - `Candidates_per_query`: Number of candidates for each query video.
- **Score Calculation:**
  - Use `_global_threshold_knn_search` to compute scores and return candidates.
  - **Metric:** `faiss.METRIC_INNER_PRODUCT` (cosine similarity), ranging from -1 to 1, with higher values being better.
  - Perform a range search (initialize `initial_radius` as a small value) to find scores within a specific radius, then sort pairs by score.
  - If results exceed `global_k`, adjust the radius until the result count is below `global_k`.
  - **Output:** A list with columns: `query_id`, `ref_id`, `query_timestamps`, `ref_timestamps`, and score. Return top scores equal to `candidates_per_query`.
- **Average Precision Calculation:**
  - Input: Ground truth and candidate (predicted) results.

- Merge the predicted results with ground truth (`how=left`). If a pair exists in the ground truth, assign a value of 1; otherwise, 0.
- Use the `average_precision_score` function from `scikit-learn` to compute AP.
- Adjust the score by multiplying AP with True positive/len(ground truth) to ensure no true positive points are missed.