Multimedia Information Retrieval: Video Browsing

Nirali Nirad Shah
Dept. of Computer Science, New York University
New York, U.S.A.
nns271@nyu.edu

Samuel Richard Smith
Dept. of Computer Science, New York University
New York, U.S.A.
srs848@nyu.edu

Ya-Ting Debby Hsu
Dept. of Computer Science, New York University
New York, U.S.A.
yth236@nyu.edu

***Abstract*—**

With advent of internet, there has been tremendous surge in search for videos. Research suggests that users spend a lot of time browsing: viewing part of one video after another while watching only a few of the videos to its completion. This is termed as seek, a special form of browsing - repeating partial viewing of the same video. This impacts the network bandwidth resulting in streaming issues with the browser. We propose to devise a system where the user would be able to view pre selected video sequence which would contain parts required by the user. This method of video browsing would be significant improvement over the traditional video browsing in terms of efficiency and user friendliness.

Keywords—video processing, frames, visual recognition, image indexing, visual content tags

I. INTRODUCTION

There are millions of videos uploaded on Internet as a result of entertainment, publishing, media and education. During the past decade there has been fast advancing research for multimedia information retrieval which includes text, hypertext, audio, image, graphics, and user interaction. In order to develop multimedia retrieval, researchers have leveraged existing methodologies of information retrieval including databases, image processing, data mining, and statistical modelling. Challenges that still exist in content based video retrieval and browsing technologies are video parsing, shot detection, scene grouping and story segmentation. In this paper, we consider a data store of YouTube videos which are scraped from internet to form corpus and used for video browsing.

Corpus also termed as Data store consists of video metadata where each video has frame data. These frames are categorized by visual recognition service of Watson generating useful visual content tags. These tags form cardinal ingredient in indexing.

Finally, the frontend server uses the classic term weighting by cosine similarity function and ranks results for particular query.

II. MOTIVATION

The emergence of web 2.0 and rise of digital media, Google brought to us YouTube introduce to us a new concept of content based information retrieval of audio and video datasets. With the rapid growth of web 2.0, video data is becoming increasingly abundant and there is continuous requirement for indexing this data in a fashion that would reduce network bandwidth and provide improved partial video browsing. An interesting problem is how bring an intelligent yet novel way of browsing videos on internet.

III. RELATED WORK

Research to portray diverse [2] fields of multimedia system roughly stated current activities residing in a multidimensional space, the axes of which can be related to different categories- System, Content, Services Usage, Implementation and Evaluation. System specified different architecture, enterprise integration, interoperation, design, security, and protocols. Content category included metadata, standards, formats, compression. Services talked about content creation, crawling, storage, browse, measurements, similarity retrieval, classification, categorization, filtering, clustering, summarization, mining, preservation, decision support, multi-modal fusion, user modeling/personalization, aesthetics ranking. Usage characterized as human-computer interface, interaction, feedback, psychology, hardware devices. Concluding with importance of diversity in multimedia information retrieval stressing (A) benchmarking related to innovation (B) developing tools v/s systems (C) following trends or using creativity and finally (D) ideal interactive research in fields on academia and industry.

Another research done in this area have described multi layer video browsing design [1] considering users view to be important where users need just to preview some video shots instead of viewing every video frames thus making the proposed system superior to the conventional frame-based browsing method from the view points of the network bandwidth, user friendliness, and/or browsing efficiency.

IV. OVERVIEW

Given the fact that there wasn't an appropriate dataset available, we had to build our own. As our target was YouTube videos, and the YouTube API did not provide a means of getting to the actual video, only the metadata, we had to build a scraper to collect our video frames. The scraper searches YouTube with a randomly selected phrase, and then begins a depth-first-traversal through the results and the related videos. Trying to create a "concentration" of results, the bank of search phrases is all related to animals, and we never go beyond a few

degrees from the results page, however we did find that the content of videos deviated very quickly.

First, the corpus was retrieved using RESTful API to be send to visual recognition service of IBM bluemix. The visual recognition service is hosted on Watson Developer Cloud that could recognize the content of images – visual concepts tag the image, detect a type hierarchy in categories of animals, automobile, retail, inventory, etc. The accuracy of recognizing frames in for the corpus we generated was about 90% allowing 250 images to be processed in a day. The visual concepts tag for frames was further used by indexer to form the inverted index and term frequency documents which are the basis for indexing.

As the corpus is continually generated, the index files obtained from indexer consists of part of data. In order to account for the data store that is already processed on cloud, the pickle files are updated by indexer at interval of 10 minutes and read by frontend servers. The frontend server then forms Inverse Document Frequency and use the classic cosine similarity algorithm to rank results for our video browsing system.

The following sections describe the flow of our project.

V. DESIGN

The project adheres to the flow as described by the architecture represented in figure 1. The main processes of the information retrieval are scraper, datasource uploader, imgur uploader, image downloader, visual recognition, indexer, frontend server, index server and datasource server.

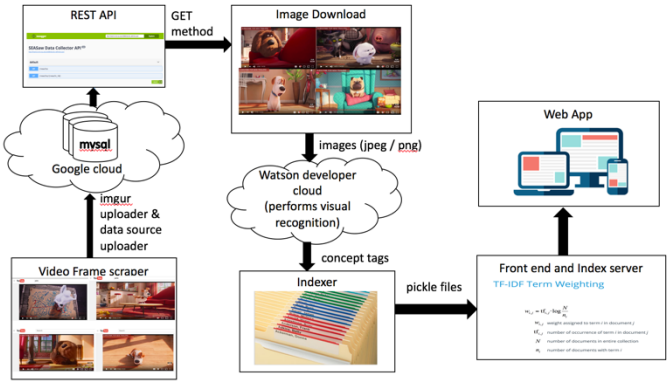

Figure 1: Video browsing architecture

The project was implemented using the classic Indexing architecture of Information Retrieval. The indexing process for video dataset is as shown in Figure 2.

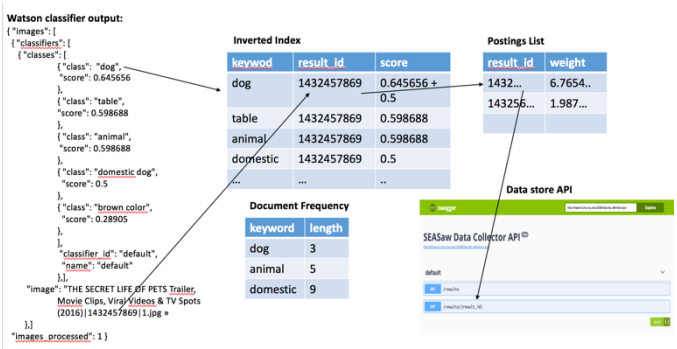

Figure 2: Indexing process

VI. IMPLEMENTATION

One of the key technical challenges of this project, and the first step in our processing pipeline, was creating the dataset from which the information retrieval system was built on.

The dataset is ultimately presented to the world using a simple RESTful API that' built on the Tornado framework for Python. When the API is running, it is also running its own documentation on swagger, which is available at /doc. The documentation is interactive, so users can tune their requests and get an immediate response.

Aside from /doc, the API has only two more accessible paths: /results, and /results/<result_id>. /results is essentially a query to get a bunch of result_ids, (which can be tuned with various parameters) and results/result_id returns the related object.

A "result" in this context, is essentially what comes out of processing one YouTube video - it contains URLs to the frame images, the timestamps that those frames occur, the YouTube title, and the URL to the video on YouTube itself. After the later step of running the visual recognition, the result also contains the related tags to the video. An illustration of Data Store API is as shown in Figure 3 and Figure 4.

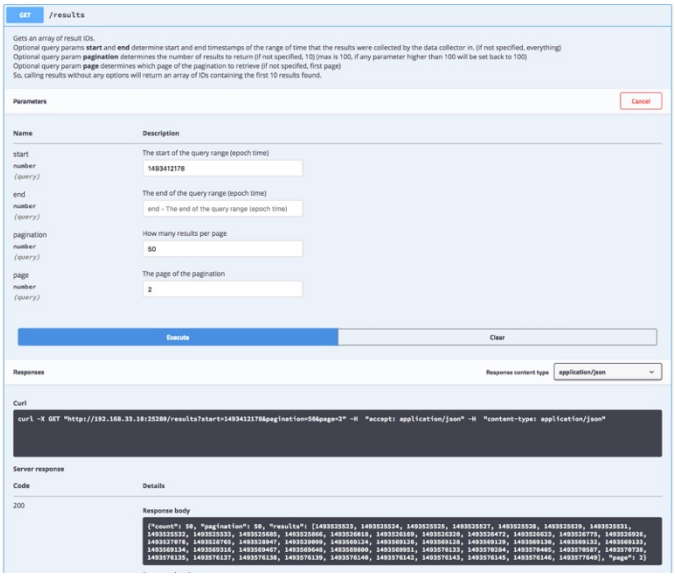

Figure 3: Data Store API (GET /result)

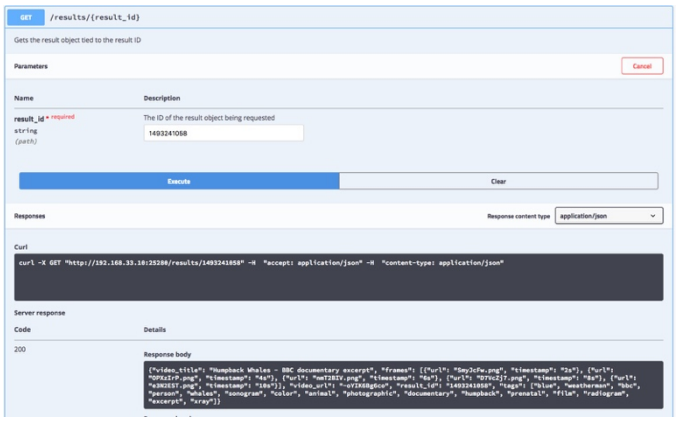

Figure 4: Data Store API (GET /results/result_id)

The results are all stored in Google Cloud SQL (mysql) [8][9], as we only take a fixed number of frames from each video (5 frames), we could have had a light and simple schema, and thus we could use a traditional SQL database and utilize its strengths in executing our primarily selection queries. We connect to the Google Cloud using pymysql and a proxy.

The frame images themselves are hosted on imgur [7], which we upload using the imgur client Python library. This presented a bottleneck - the imgur API is limited to 50 images per hour, and 1250 per month, for non-commercial purposes.

Gathering the frames is done using selenium for python, running on the Firefox web browser. Due to the fact that this was a hotly-in-development toolset, we had 3 major versions release during our project time. Being in this early phase meant that there were numerous bugs and unpredictable behaviors, so part of the technical challenge in this area was ensuring that we could recover gracefully and try again, without losing too much work.

The scraper, imgur-uploader, database-uploader, and API all run in their own processes, this allows the scraper to move on while the frames are being uploaded, and so-forth. Each phase writes its output to disk, and creates a file when done. Next phases will scan for that file before beginning its part.

Next, scheduler is run by passing database and proxy credentials to connect to Google Cloud Proxy. This scheduler is of blocking type and is invoked every 10 minutes to account for real time data generation. It also takes care of three major components namely image downloader, visual recognition and indexer.

Image downloader (a.k.a. frame downloader), invoked by scheduler, is responsible for frame data which is obtained by querying data store using REST API, described in detail in Figure 3. This downloader also keeps track of previously processed images in order to avoid duplication in inverted index. Further, the frames are queried using the API described by Figure 4, in round robin fashion. The images are downloaded in parallel by multiprocessing and are stored in temporary file created by scheduler. The format used by image downloader to save is as shown in equation 1.

<video_title>|<result_id>|<frame_number> (1)

This format is of importance to indexer while creating inverted index and postings list. The image downloader also, takes care of cleaning the title due to limitations of Watson's visual recognition service.

In order to index video frames, we used visual recognition service hosted by IBM bluemix on its Watson developer cloud. This service uses service credentials in form of API-key to authenticate and daily limit before generating results. Visual Recognition [3] service uses deep learning algorithms to identify scenes, objects, and celebrity faces in images that are uploaded to the service. The method [6] used for classify images are as delineated in table 1 below. Certain prerequisite requirements are-

A) an image passed to the service must be in either .jpeg or .png format
B) an image that is a minimum of 224 x 224 pixels for better quality results
C) an image size should be less than 2 MB
D) the max number of images in a .zip file sent to classifier is limited to 20, and limited to 5 MB.

Table 1: Classifier Method details

Parameter	**Type**	**Description**
api-key	query	(Required) API key
version	query	(Required) The release date of the version of the API you want to use. Specify dates in YYYY-MM-DD format. The current version is 2016-05-20
images_file	multipart/form-data	(Required) The image file (.jpg, or .png) or compressed (.zip) file of images to classify.
Parameters	multipart/form-data	A JSON file containing input parameters. Input parameters can include: **url** - A string with the image URL to analyze. **classifier_ids** - An array of classifier IDs to classify the images against. **owners** - An array with the value(s) "IBM" and/or "me" to specify which classifiers to run. **threshold** - A floating point value that specifies the minimum score a class must have to be displayed in the response

The response [6] of the classifier is dictionary which contains elements characterized in table 2.

Table 2: Response from classifier

Name	Description
images	An array of images classified
classifiers	An array of the classifiers detected in the images
classes	An array of classes within a classifier
class	The name of the class identified in the image
score	The score of a class identified in the image. Scores range from 0-1, with a higher score indicating greater correlation
classifier_id	The ID of a classifier identified in the image
name	The name of the classifier identified in the image
image	The relative path of the image file. This is omitted if the image was passed by URL
source_url	The source URL of the image, before any redirects. This is omitted if the image was uploaded
resolved_url	The fully-resolved URL of the image, after redirects are followed. This is omitted if the image was uploaded
images_processed	The number of images processed by the API call
error	An object containing information about what might have caused a failure, such as an image that is too large. This is omitted if there is no error or warning
error_id	A codified string ID for a specific error. For example, "limit_exceeded"
description	A human-readable error description. For example, "File size limit (1MB) exceeded"
warnings	An array containing information about what might cause the output to be less than optimal. For example, a call with a corrupt .zip file and a list of image URLs will still complete but will not have the expected output. This is omitted if there is no warning

The response from visual recognition service is passed to indexer in format illustrated in figure 5. The indexer [4] iterates over each response, case folds the text, and tokenizes using NLTK's word_tokenize. It then finds the index partition (and associated inverted index) corresponding to the current result_id (extracted from "image" key in response) by calculating result_id modulo index_partitions implying the use of document partition rather than term partition. For each term in the text, it adds the current result_id and the term's frequency in the image to the inverted index's postings list for the term. A bonus of 10 is given to terms that appear in the title. The indexer also keeps track of the result_id frequency (document frequency) of each term. Finally, it writes the result_id along with its associated terms to database. On successful insertion, it appends out a pickled inverted index for each index partition. It also writes out a pickled file for videos processed.

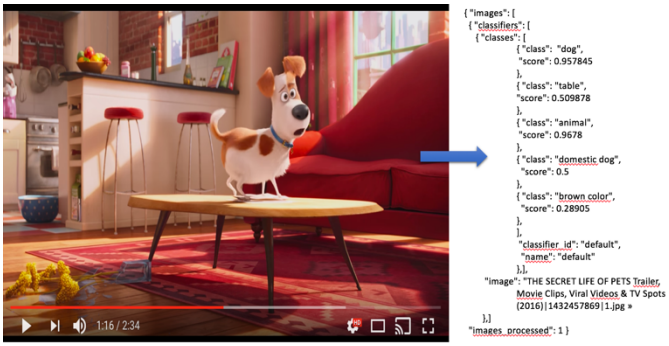

Figure 5: Watson's classifier output

The frontend server loads several pickled files when a search is triggered. It first loads the Inverted Index files into a list. Since there are different partitions of the server, this list is then merged to form a dictionary where each term, as key, only appears once in the inverted index and the value is a dictionary of result ids and the term frequency of the term in the associated result ids. Next, frontend server loads the Processed Videos file and forms a list. The purpose of this list is for the calculation of IDF. The IDF is calculated using the last part of the TF-IDF formula in figure 6.

$$w_{i,j} = \mathrm{tf}_{i,j} \cdot \log \frac{N}{n_i}$$

$w_{i,j}$ weight assigned to term *i* in document *j*

$\mathrm{tf}_{i,j}$ number of occurrence of term *i* in document *j*

N number of documents in entire collection

n_i number of documents with term *i*

Figure 6: Term weighting (TF-IDF)

The frontend server is now ready to process queries. The query terms are fetched and split into query_vector which is then used to form a dictionary doc_vec_dict. For each query terms, the server traverses through the inverted index (INV) to get a dictionary of result_ids and its associated term frequency (TF) from it. The TF is then used to calculate the TF-IDF value using the formula in figure 6 and the values are saved in the doc_vec_dict with result_id as key and tf-idf for the value. The keys are then retrieved from doc_vec_dict to form a new list doc_id.

If the doc_id is empty, then the server determines that the search yields no result and displays "Results not found" message on the web page. If there are matching results, the

server will sort the results with tf-idf value using the argsort() function from linear_kernel from sklearn.metrics.pairwise and the list of result_ids is saved into best_doc_ids.

The frontend server then requests results data from the datasource server. It fetches the video title, video url, tags associated with the video and the frames as well as timestamp from the datasource server and these information is created as an instance of the Result case, helps in straight forward parsing of results.

Finally, the server uses generateResultRows() functions to generate the html code and pass it to template.html and renders it for the user to view on the page.

The main challenges and drawbacks faced while building this project are many fold. The following list is a comprehensive summary:

- A part of the Selenium pipeline, the geckodriver, is under heavy development and has a number of bugs and inconsistencies, meaning that the scraper had to be tolerant of seemingly-random failures.
- Geckodriver froze on launching if trying to run Firefox with extensions. As such, we could not prevent adverts from running on YouTube, which caused inconsistencies with our data.
- Watson has its own limitation of only processing about 250 images a day which meant we would have to wait long to process around 300 videos. It has limitations of processing images that have titles without emoji's. Watson fails to process images that do not provide quality resolution. Major drawback was that it could not process images in parallel implying waiting for at least 5 seconds before forming a new connection.
- Image downloader uses urllib.request.urlretrieve functionality of python to save images. Existing bug with urllib was that it failed to retrieve request whose title had emoji's unicoded. This was fixed as stated in https://github.com/aio-libs/aiohttp/issues/207 but it still exists.

VII. Results And Obervations

When search for an animal or pet, the application would give us results in form of tiles. These tiles are sorted by frontend server where each result consists of title, URL of the video, frames of the videos and a small subset of visual tags. The frames are displayed such that when the user selects a frame, frontend directs it to the URL and starts playing from the timestamp where the frame was present in the video.

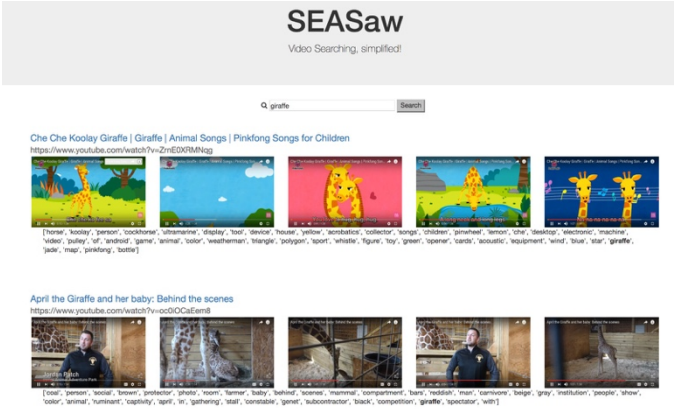

Figure 7: Results for "giraffe" on web app

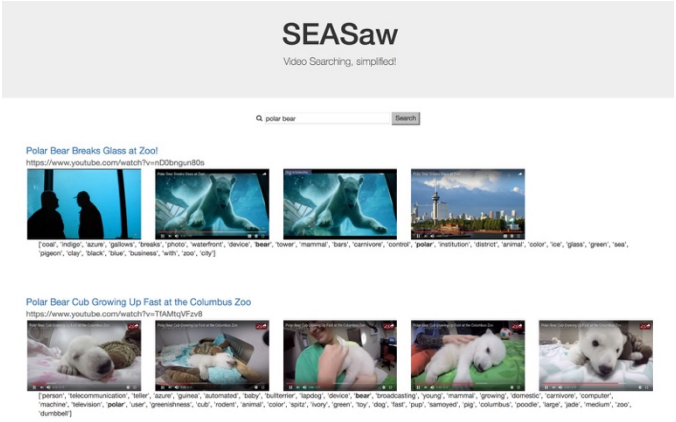

Figure 8: Results for "polar bear" on web app

There might be certain discrepancies in the results of certain terms because the scraper doesn't account for ads present in a video. Also, the quality of the frames is not considered by scraper due to which Watson service would fail. Another, shortcoming due to which our results may look awry is that Watson service accounts for 95% accuracy.

VIII. Future Work

One of the most obvious improvements we could make to this project will be to move into the commercial tiers of the APIs that we used, so that we can process a far larger volume of data. Also, later down the line, when geckodriver is more stable, we can run again and build a dataset that doesn't have the integrity issues caused by the YouTube commercials.

Beyond that, we would start taking larger samples from each video - currently we're taking 5 frames per video, in the future we could scale up to, say, a frame per minute. A lot of potential context from the video is missing with such a few number of frames.

Moving into another feature to process, we would like to start analyzing the audio contents of videos (though this may posit more legal issues). Many videos on YouTube are usually just a person talking about a certain subject. Analyzing any text

that may appear may will help with indexing videos that are uploaded lectures.

IX. Conclusion

Due to insufficient video dataset present on Internet, we had to built our own data by scraping which increased man number of days to complete our project. Also, watson processes framed sequentially as it doesn't support multiple connections to be made. The information retrieval system we built fails in certain cases when the visual concept tags provided by watson are not completely accurate.

Acknowledgment

The authors would like to thank Prof. Math Doherty for encouraging us to work on Multimedia information retrieval system. We would like to thank Courant Institute of Mathematical Sciences, Computer Science Department for providing computing resources. The opinions expressed in this article are their personal views.

References

[1] Herng-Yow Chen and Ja-Ling Wu. Communication & Multimedia Lab, Department of Computer Science and Information Engineering National Taiwan University, Taipei, Taiwan, R.O.C. "A MULTI-LAYER VIDEO BROWSING SYSTEM" *IEEE Transactions on Consumer Electronics, Vol. 41, No. 3, AUGUST 1995*

[2] James Z. Wang1 (Chair), Nozha Boujemaa2, Alberto Del Bimbo3, Donald Geman4, Alexander G. Hauptmann5, and Jelena Tesic. The Pennsylvania State University, University Park, PA, USA 2 INRIA, Rocquencourt, France 3 University of Florence, Florence, Italy 4 Johns Hopkins University, Baltimore, MD, USA 5 Carnegie Mellon University, Pittsburgh, PA, USA 6 IBM T. J. Watson Research Center, Hawthorne, NY, USA. "Diversity in Multimedia Information Retrieval Research" *MIR'06, October 26–27, 2006, Santa Barbara, California, USA. Copyright 2006 ACM 1-59593-495-2/06/0010*

[3] Visual Recognition service hosted by Watson developer cloud. *https://www.ibm.com/watson/developercloud/visual-recognition.html*

[4] Indexing and Retrieval from *http://cs.nyu.edu/courses/spring17/CSCI-GA.3033-006/assignments.html*

[5] Tornado Documentation for implementing frontend and index servers from *http://www.tornadoweb.org/en/stable/tcpclient.html*

[6] Visual Recognition documentation for classifier method from *https://www.ibm.com/watson/developercloud/doc/visual-recognition/getting-started.html#step-2-classifying-an-image*

[7] Examples provided and readme for image uploading to imgur from *https://github.com/Imgur/imgurpython*

[8] Google cloud database up and running from *https://cloud.google.com/sql/docs/mysql/quickstart*

[9] Connect the database to application from *https://cloud.google.com/sql/docs/mysql/connect-external-app*

[10] Setting up selenium from *http://selenium-python.readthedocs.io/getting-started.html*

[11] Code References: Creating vagrant file from *https://www.vagrantup.com/docs/index.html*

[12] Code References: Travis ci from *https://docs.travis-ci.com/user/languages/python/*