FINAL REPORT

Larissa Djoufack Basso, Rachell Kim, Weiming Long
CSCI 4243 Senior Design
Prof. Pablo Frank-Bolton, Prof. Rahul Simha, Elyse Nicolas, Chris Toombs.
May 5, 2020

Section 1 — Work Distribution

I. Larissa Djoufack Basso

A. Implemented a code to collect data from s3

B. Implemented the machine learning algorithms of the project. This by implementing a video tagging feature using machine learning algorithms and storing it in mongoDB.

C. Implemented the search functionality to enable users to search for words not present among the video labels previously stored in mongoDB.

II. Rachell Kim

A. Built the initial skeleton web-application using Angular and Flask frameworks

B. Implemented most frontend features and functionalities, including the full search and upload functionalities

C. Assisted database management for the S3 and MongoDB

III. Weiming Long

A. Build the video processing part using FFMpeg to extract audio from the video file and use Google Cloud Speech-to-Text API to do the transcription.

B. Managed file cloud storage for video files, audio files, transcript, and timestamp table.

C. Implemented Categories feature on the website

Section 2 — Non-technical Overview

As students who often engage in the use of multimedia for educational purposes, we recognize that the greatest challenge in learning through recorded instructional videos is the task of finding the exact location of relevant topics. Our system, called the QA Classifier, not only categorizes video material by subject, but also provides time frames in which relevant keywords and topics are spoken. The idealistic instructional platform for recorded video and audio materials should not only return relevant results to the user, but also provide precise times in which certain subject matters are discussed. This is precisely what the QA Classifier achieves. Because office hours typically take place in spaces that are inaccessible to students who cannot physically attend, instructors are often met with the tedious task of reiterating explanations for multiple students within the same class. Additionally, lectures and review sessions tend to happen in lengths of time that are not considered concise. To mitigate these issues, the QA Classifier was designed to allow users to watch recorded instructional videos with the option of searching for spoken topics using keywords.

To use the application, users may simply navigate to the homepage and search for a keyword or subject they wish to find within a video. The system will then return to the user all videos containing the queried keyword, along with a list of time frames in which the keyword was spoken. Users also have the option of uploading videos into the system by navigating to the Upload Panel of the application. Moreover, the QA Classifier includes a Category feature which allows users to select and view all videos within a specific category. In short, this system serves as an educational tool for accelerated learning using video mediums by providing the means to look up specific information and references within audio content.

Section 3 — Link to the Project Website

I. Link: https://weiminglong.github.io/QA-Classifier/

II. Contents:

A. Screencast of the final presentation

B. Pictures of each team member including a short bio

C. Links to previous writing assignments as well as the Final Report

Section 4 — Libraries, Packages, and APIs

I. Frontend dependencies
   A. Angular CLI
   B. npm
   C. Node.js

II. Backend dependencies*
   A. Flask
   B. Anaconda
   C. PyMongo
   D. Boto3
   E. GCloud
   F. SciKit
   G. NLTK

*The complete list can be found in the environment.yml file within the repository

Section 5 — Technical Overview

Our system, the QA Classifier, is a web-application that parses and classifies segments of recorded instructional videos. With this system, users may search for spoken keywords and topics within uploaded videos using a search bar from our web-interface. Once an instructional

video is uploaded into the application, the system will then parse it's contents and automatically tag relevant keywords to certain time frames. The video, the keywords, and time frames associated with the keywords within the video are then stored into the database, later to be retrieved when users query for the tags associated with that video. The QA Classifier is built on top of two application frameworks: Angular and Flask. Angular is used to create the user interface and provides management to the frontend, while Flask is used to run service requests and handle HTTP calls from users.

The workflow of our system is as follows. First, to use the system, users must first upload instructional videos through the frontend application. The frontend receives the video and sends the video file, along with its metadata, to the Flask backend application, where it is handled for processing. Our application uses FFMPEG to format received videos and transcribes the extracted audio file using the Google Speech to Text API. Videos longer than three minutes require upload to Google Cloud Storage for processing. Once this is finished, two output files will have been made: a text file containing the transcribed audio and a csv file listing each word along with the time it appears within the video. These two files are stored in the Amazon S3 bucket, along with the original video file. The natural language processing portion of our application then performs the TF-IDF, Wu-Palmer similarity, and word embedding algorithms to determine the top five keywords of each video within the system, readjusting previous values for other videos if they have changed. Then, the keywords, timeframes, and link to the video within S3 are stored in the Mongo database. This completes the upload process. Once this is done, users may search keywords within the application to search videos by their audio files, and use the keywords to jump around within videos.

Section 6 — If I Had to Do This Again

Larissa:

If I had to do this again, I would have used a model based-approach while writing the functions to map words and their respective start time and videos. It was extremely time consuming to work on this section, because I had to ensure that I tested all of the different edge cases and spent a lot of time debugging and testing my implementation. I did not have much python programming knowledge and at the beginning I did not fully take advantage of python libraries. Further, I wish I had a better understanding of different Machine Learning Algorithms as I do now, especially neural networks and implemented it for our project mainly on the TFIDF portion. Further, word embedding is a data hungry algorithm. In order to return similar words a user searches especially for words that are not even in the videos, using pre-trained word embedding models would have yielded more similar contextual words. Given what I have learned in my Machine Learning class, there are other features that I could have implemented here, such as sentence or multi-word search. In order to implement it I could have gotten vectors from the model returned by the word2vec algorithm and returned a vector average for all the words in the sentence query the user passed in.

Nonetheless, I am very proud of what I have accomplished and the end results. It was a great learning process and I enjoyed every moment of it. Through this process, I had the opportunities to learn about other machine learning algorithms whether it be via the books or blogs that I read.

Rachell:

If I had to do this again, I would thoroughly research the limitations of all our libraries and APIs ahead of time to prevent reimplementing certain parts of the system. One of the most challenging parts of designing the QA Classifier was dealing with the uncertainty of what was possible and what was not. For example, one of our core functionalities required the Google Speech to Text API. However, it was only deep into the Fall semester when our team had realized that this API requires the use of Google Cloud Storage for any video longer than 3 minutes. Had we discovered this earlier, I believe we may have made different decisions in terms of system architecture. The same was true for many of the natural language processing libraries that we had initially employed to build our application.

Regardless of the setbacks, I believe this journey was a great learning process and I am grateful for the experience. Even though it took us a very long time to figure out what would bring the system together, I think figuring it out through trial and error was a necessary part of the process. Although there are many design choices that I would have liked to have done differently in hindsight, I am satisfied with the overall product.

Weiming:

There were many mistakes made in my part of the project. Looking back from the beginning to the end, I learnt a lot and can help me to prevent those mistakes happening again in the future. The biggest mistake I made is that I didn’t realize the requirement of using Google Cloud storage bucket to store large audio files in order for them to be transcribed. In the beginning I was testing the Speech-to-Text API from Google Cloud using small audio files

stored in my local computer. Those audio files I used were mostly one or two minutes long and all less than 10 MB in size. The transcription worked perfectly and I was confident in integrating it into the whole system. I overlooked the scenario that the educational videos can be 30 minutes or hours long and audio parts extracted from them would be huge in file size. It was until the mid phase of our project when we put everyone's part together and tried out large videos, then the error of file size exceeds limit occured. I was panicking that the previous perfect transcription just didn't work at all. Even though the mistake was found not too late, it took a lot of work to change a big part of the system's structure. For audio files that are larger than 10 MB, they have to be uploaded to Google Cloud Storage bucket first in order for them to be transcribed. I didn't realize when transcribing small audio files, the process is still uploading them to Google Cloud first but stored in a "free" storage bucket. I have to create my own storage bucket to store large files first. Then I modified the whole process to upload the audio file right after it was extracted from the video file, then retrieve the URI and pass it in as an argument for the audio transcription. This process added a whole new database for our system structure besides AWS S3 and MongoDB. Making this big change will make our system more complicated thus more potential errors. Therefore I should scrutinize every API thoroughly in the future to prevent mistakes like this.

One improvement that I can make is to try out different solutions of audio to text APIs. Even though Google Cloud solution is so popular that we can see on YouTube, Google Translate, and Google Home etc, there are still many available solutions from AWS, Azure, and Oracle. When developing for larger projects, the pricing part needs to be calculated accurately. I could have implemented every one of them and compare their performance side by side. The

transcription time is crucial in some usage. The accuracy of complicated terminology also needs to be checked carefully. Another improvement is to make the system structure cleaner by removing the AWS S3 part and store everything in Google Cloud Storage. The video files, output text files could all be stored in Google Cloud Storage in one place if I found the limit of local audio file size for transcription earlier. The design of system structure is important and making changes during the development is troublesome and should be avoided.

Section 7 — Follow-Up Projects

I. How to set up the project

   A. Download the zip file from the Github repository

   B. Backend

      1. Create a conda environment for the backend dependencies and activate
      2. Install dependencies using the environment.yml file found within the repository
      3. Create a MongoDB account and AWS account
      4. Configure the AWS S3 bucket access and Google Cloud access in the console and replace authentication strings (*note: must use the same database and collection names)
         a) app.py for MongoDB
         b) auto.py for Google Cloud
      5. Configure `export FLASK_APP=app.py`

   C. Frontend

1. Install node.js and npm
2. Install Angular CLI and Angular Material

II. Issues and pitfalls

A. TF-IDF algorithm: The current algorithm requires all documents to be read upon each upload. Ideally, this should be changed to prevent large overhead of processing all documents upon uploading a video.

B. Routing: There are some unresolved routing issues when attempting to “click to go back” (i.e.: video no longer loads if the user is on the viewer panel)

C. Upload feedback: Snackbar feedback may not show up sometimes if the upload panel gets automatically refreshed.

D. Multi-word search: The current implementation does not allow multi-word queries in searches. This is not an issue per se, but it could be improved to include this feature.

III. Future ideas

A. Our initial proposal for the QA Classifier included a feature which includes the information on the web. Although the QA Classifier was originally meant for “local” learning (i.e. meaning videos uploaded into the system would be used by the university students), it would also be helpful if the system could provide resources similar to the ones in the application. If all videos within the QA Classifier are lecture videos recorded by instructors, maybe provide a sidebar which lists similar instructional videos on the web. This will most likely need to incorporate computer vision.

B. Upgrading the original TF-IDF algorithm or changing the algorithms to use something more efficient may also be a good start to an extended project.

C. Implement sentence search functionality will be a great extension to the project as well.