Image to Speech

via

Canny Text Detector

Final Project Report
ECE 420

Eric Mysliwiec (emysliw2)
Martin Lamping (mdl3)

TA: Grant Greenberg
Section: ABC

Introduction

Computer vision is a new and exciting field that encompasses many concepts central to digital signal processing. Text scene detection is a major component within the computer vision field where exciting advancements have been made recently. One of these advancements was a new and novel way to utilize a modified canny edge detection algorithm [2]. With this idea, we implemented a text detecting algorithm that was central to the "Image text to speech" application. The image to text to speech app first detected the text in an image using an altered version of text detection from the Canny Text Detection detailed by CCho, Hojin, Myungchul Sung, and Bongjin Jun at the IEEE Pattern Recognition Conference in 2016. It then sent those detected regions to an OCR function provided by Google's Text API. This function would recognize words and return a string to a text to speech function, available since Android API version 4.0. We had varying levels of success but by the end of project we had a working detection app, that would recognize the text and read it back. The following report first provides a description of the original text detection app described by Cho et al. It then introduces other comparable apps on the market and our approach to solving this problem.

Research Review

One such application of a modified Canny edge detection algorithm is text detection. Text detection implementations have been around for quite some time and text detection APIs have become a common utility in recent years [1]. The implementation investigated here will be the Canny Text Detection algorithm first described by Cho, Hojin, Myungchul Sung, and Bongjin Jun at the IEEE Pattern Recognition Conference in 2016. [2]

Algorithm:

The algorithm consists of two main combined processes:

1. Character extraction ER (extremal regions)
2. Further classification and analysis via modified Canny edge detection [2]

In the sections that follow, we will detail the Canny Text Detection algorithm from Cho et al.

Background Info on the Canny Text Detection Algorithm

ER Character Extraction

In the canny text detection algorithm, the first process is to extract character candidates using extremal regions (ER) and form a component tree. An ER is defined as "a set of connected pixels in an image whose intensity values are higher than its outer boundary pixels."[1] This technique is applied over single image channels, therefore if we apply to a color image, the ER must be applied to R, G and B channels. These text candidates can be found using connected components labeling within a particular image channel. Connected component labeling is grouping pixels based on their shared intensity values and close spatial proximities [3]. It may be best to resolve character candidates using the intensity value of normalized image data.

This will reduce the overall computational of the algorithm and help reduce latency. A mathematical description of ER's can be found here:

$$R_t = \{x | I(x) > I(y); \quad \forall x \in R_t, \ \forall y \in B(R_t)\} \qquad (5)$$

A component tree will be used to store the extracted character candidates based on their intensity values. The component tree will ensure efficient data management necessary for tracking and real time usage. [4]

Non-maximum suppression

Non-maximum suppression uses the gradient magnitude and directional data found in the previous step to further classify edge pixels. The image is scanned and each pixel is compared to its neighboring pixels. In text detection, there is a large probability that text candidates will be flagged over multiple thresholds. Therefore a bounding rectangle is needed to group all areas under which only one channel will be considered [2]. This is to help ensure there are no multiple or repeating ERs.

Double threshold

Using a similar method to our pitch detection lab, this part uses a min and max threshold to characterize the edge pixel's gradient values. For text detection, the app needs to be trained in order to correctly determine character candidate validity. The threshold training values are selected to ensure accuracy and were intrinsic parameters to the trainer. Mean local binary pattern (MLBP) is then used to ensure rotational and illumination variations in character detection. Centered at any given pixel, the average value of a 3x3 tile of image pixels, is first calculated. This calculated average is then compared to all pixels except for the center pixel. The value of the pixel is set to 1 if the pixel value is larger than the average value, and set to 0 otherwise. After this value has undergone the appropriate scaling a clockwise encoding (the local binary pattern) must been encoded as the final MLBP value for the central pixel [2]. An example for MLBP is shown below:

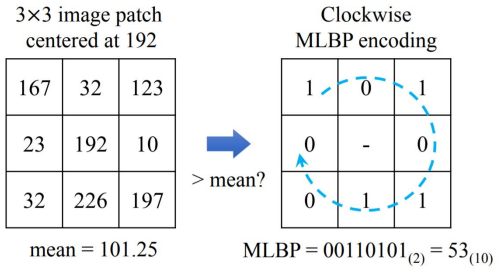

Figure 1: Shows and example of MLBP pixel encoding [9][2]

Hysteresis

Double threshold attempts to reduce the image to only text regions, however, some non-text regions still remain. Regions that are in between are considered non-text if there is no connecting part that goes above the maximum threshold and are considered text if they are connected to a segment that goes above the max threshold [3]. For text detection, neighboring strong and weak character candidates are checked against one another. If they resolve four text localization criterion then we change the weak character candidate to a strong character candidate [2].

1. They are spatially close enough to be considered in the same phrase
2. The font size is close enough to be considered the same
3. The color channel values are similar: each channel has max difference of 25
4. The stroke widths are similar.

Any weak character candidates that do not resolve these four criterion will be discarded or classified as a no-text [2].

Software implementation

Initially the text detection algorithm was prototyped in python. Starting in python allowed us to debug more efficiently and easily convert the working code into java to be applied on android similarly to lab 6. After the prototyping phase we attempted to port the lab into Android studio using C++. We were quite successful in early stages of this process however we could properly for the ERFilter in C++. After many tries it was decided to abandon the C++ version of the app and start a new version using solely Java. Java proved to be a fairly simple language to develop with and we soon produced our algorithm with good results.

Other Approaches from Existing Apps:

There are many existing apps on the market that attempt to provide novel solutions to this issue. Most are marketed as solutions to communication concerns if/when the end user finds themselves in a different country. They not only will detect the text but they will also translate the language as well. The App Store alone has many different optical character detection (OCR) apps that will detect and recognize text in scene images and turn it into a text (.txt) file. The novel idea associated with our app is that it will automatically read the detected and recognized text back to the user. In the future, we would like to expand this app to also perform language translations as an intermediate step. This easy addition to our apps functionality would increase its marketability immensely.

Technical Description of Team 21's Approach

Preprocessing Techniques

A few different preprocessing techniques were used in an attempt to increase the accuracy of the detection algorithm. Image preprocessing occurred prior to the image being sent through

our text detection algorithm. Because this algorithm uses pixel contrast to find edges of characters three major preprocessing techniques were tried:

1) Contrast Enhancement
2) Sharpening
3) Smoothing

There were varying levels of success with each, however it was found that the sharpening was the most beneficial. Sharpening produced marginally better results than not sharpening at all. Both contrast enhancement and smoothing produced worse results and in some cases made the detection rate reduce to almost nothing.

Text Detection Algorithm

We used an altered version of the approach by Cho, Hojin, Myungchul Sung, and Bongjin Jun. [2] Having only three weeks, our team of two had a lot to complete in such a short amount of time. We knew that if we could accurately detect the text, that the OCR function provided by Google's Text API should be able to recognize the text with a high success rate. Therefore, we focused our efforts on text detection.

The final implementation of the text detection algorithm includes the following:

1) ER Character extraction
2) Non-maximum Suppression
3) Double Threshold without MLBP Encoding

Note: these processes are described in detail in the sections above.

Due to time restrictions we were unable to implement MLBP Encoding so the final app did best when the characters were oriented horizontal. If the characters were slanted too far, the app would struggle to detect them. Future iterations of the app will attempt to integrate MLBP encoding into the overall algorithm. We also only read the RGB (Red-Green-Blue) color channels when we detect the ER regions. We needed to reduce these channels do decrease the runtime of the app. Future versions of the app will try to optimize this runtime even further while also trying to increase the accuracy.

Optimization: Detection Accuracy vs Runtime Considerations

To reduce runtime, we tested each channel to determine which one contained the most useful information for text detection. The imread function provides us with red, green, blue, luminance and gradient channels. Of these channels, it was found that sufficient text detection could be completed using only the RGB channels. In attempts to improve runtime even further, we tried gray-scaling the image, however, this gave us many problems and did not make for effective text detection.

Post Text Detection

After the text had been detected, it was passed to the OCR function from Google's Text Recognizer API which recognized the text and converted it to a string. The string associated with each specific region was then passed, one by one, to the text to speech function. It was noticed that without adding a delay between these the text to speech (Android Speech - Text to Speech) would produce an unintelligible sound. Once we put a delay between each regions playback, we produced something that could be easily heard and comprehended. In further iterations we would like to implement a better phrasing technique to make the playback sound more eloquent. Right now the playback is comprehensible but it sounds choppy and robotic.

Block Diagrams

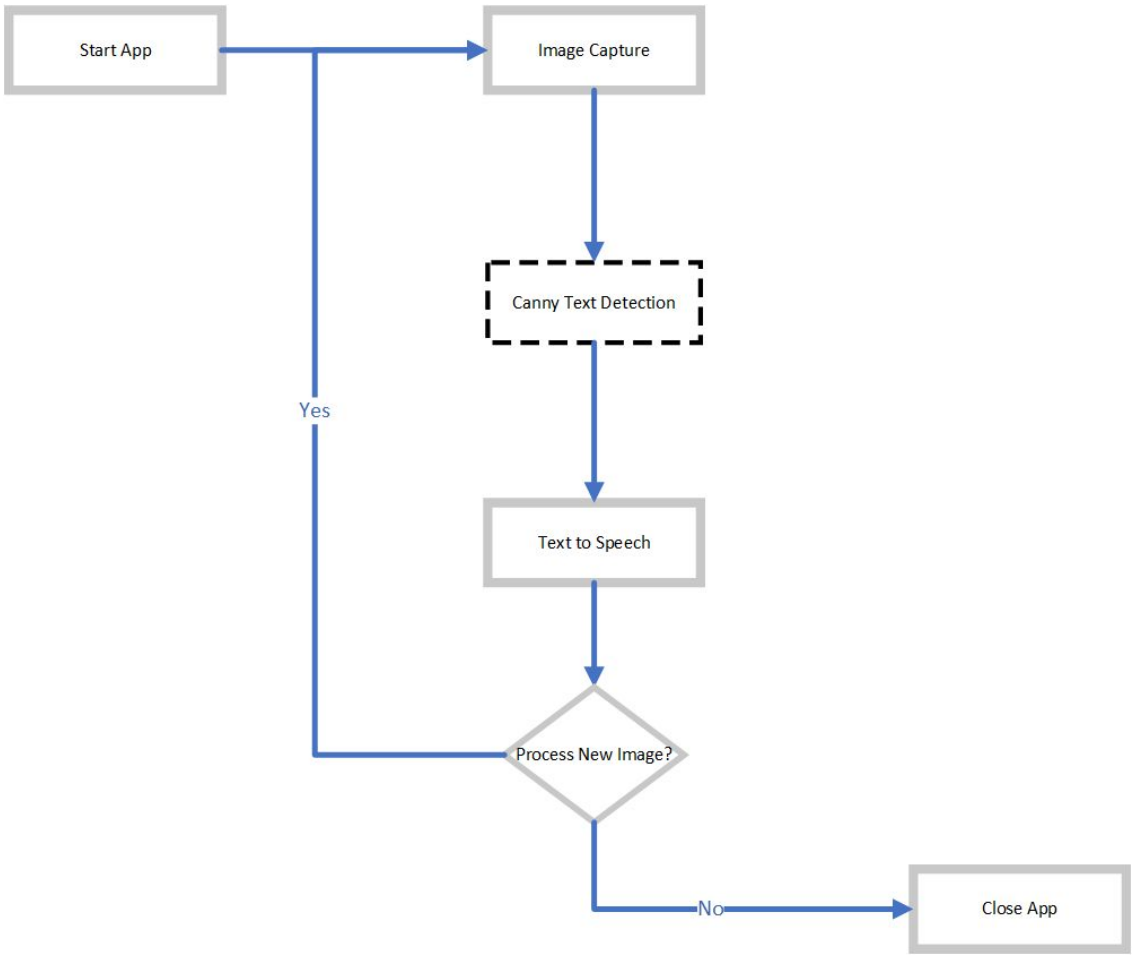

Figure 2: Software block diagram

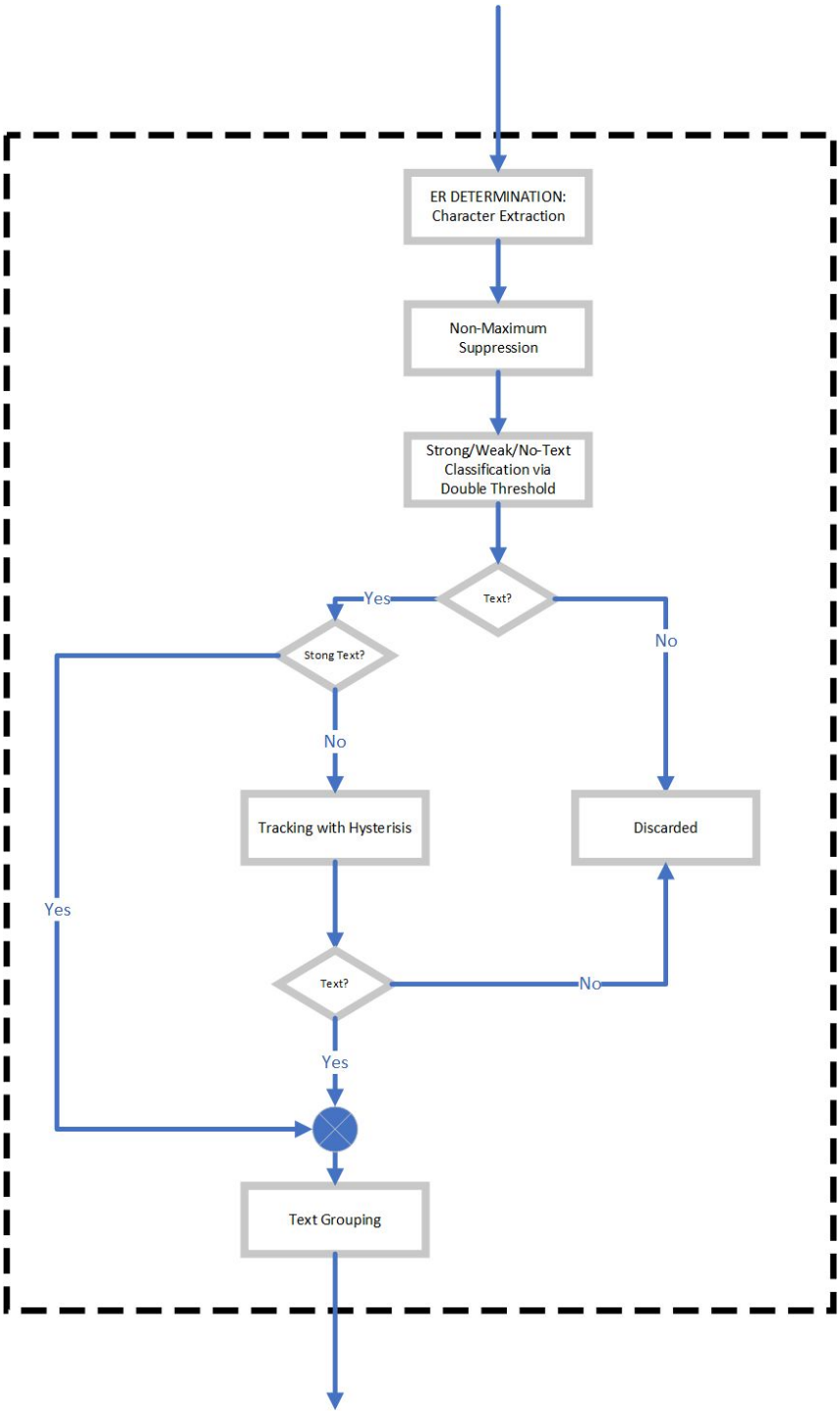

Figure 3: Canny Text Detection algorithm [2]

Results

The images shown in Figure 4 and Figure 5 are the results of the algorithm's functionality. Figure 4 is used as a negative test case to show that the algorithm will not get stumped by ER regions that could be interpreted as text. Figure 5 is used as a positive test case to show how

effectively the algorithm can separate text from other background features. It is apparent that the algorithm does not give any false positives, however, it is prone to giving false negatives. The algorithm is consistently effective when the text color has high contrast with the foreground, but is not effective when there is low contrast between the text and the foreground. This can be seen in Figure 5 by looking at the words “STARTED” and “HERE”. The ending letters were incorrectly identified as non-text by the algorithm. While the text identification works to user satisfaction, the text recognition struggles to correctly identify words. Another problem with our algorithm is its undesirable runtime. Initially it was taking over two minutes to finish, however, after identifying the most effective channels, the runtime was cut in half and can now run in 45 to 60 seconds. This runtime is still not ideal but is much improved from the original runtime.

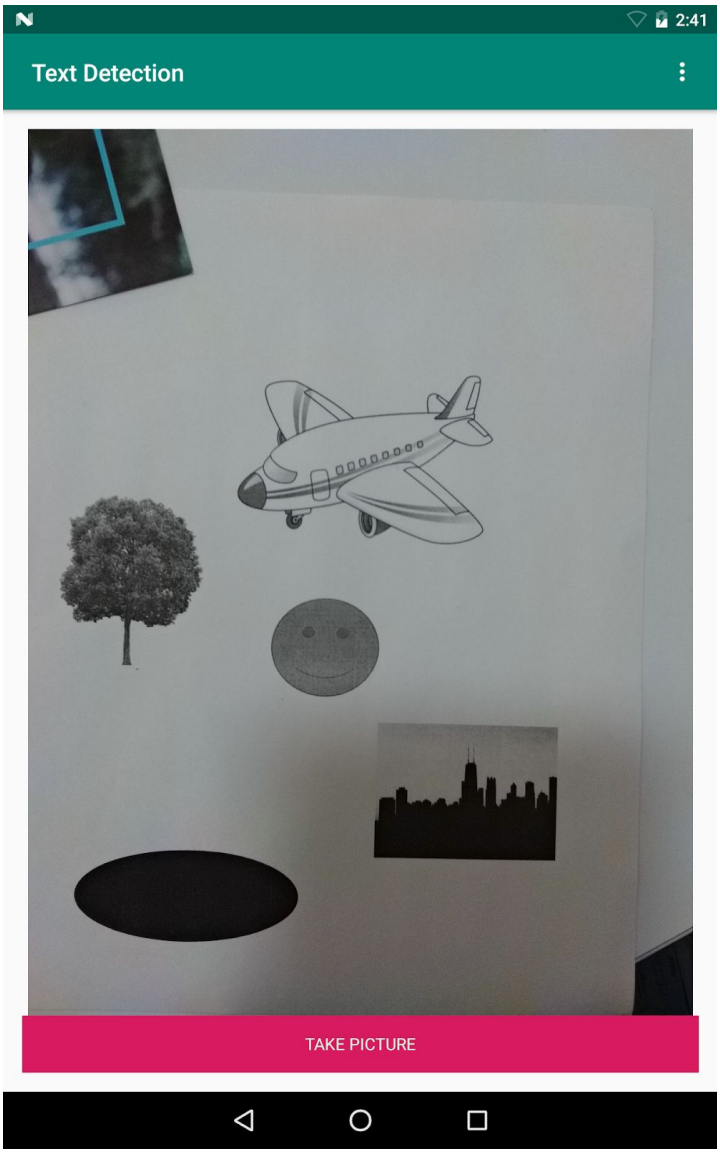

Figure 4: Negative Test Case

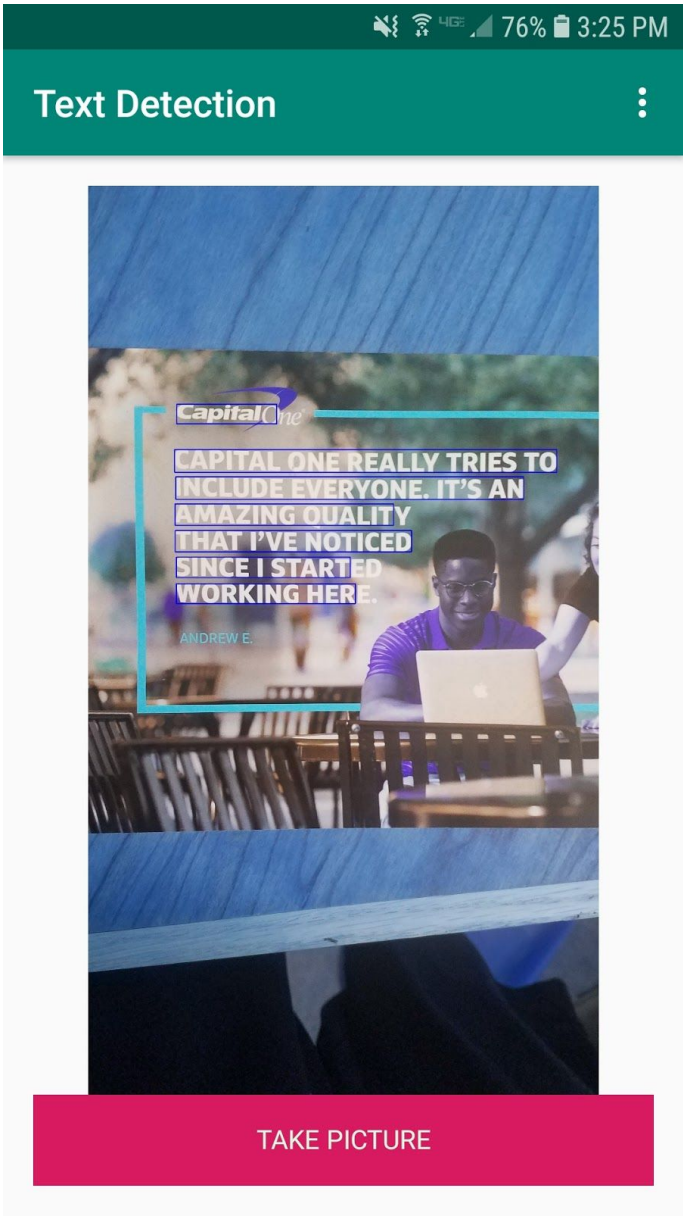

Figure 5: Positive Test Case

Conclusions

The resulting application was a successful implementation of our algorithm. We promised an app that could take an image, use the canny text detection algorithm to detect text. Then pass those regions to an OCR function that would recognize the text and convert to a string. Finally our app played back the text, through the speakers (or headphone) to the user. We therefore successfully completed all deliverables. Most of the planned implementation details went smoothly, however, our original plan to use C++ was eventually dropped due to unresolvable errors. This allowed us to meet our reach goal of text recognition and text to speech.

References

[1] P. Kinlan, “Detecting text in an image on the web in real-time,” *Detecting text in an image on the web in real-time - Modern Web Development & Tales of a Developer Advocate by Paul Kinlan*. [Online]. Available: https://paul.kinlan.me/detecting-text-in-an-image/. [Accessed: 04-Mar-2019].

[2] Cho, Hojin, Myungchul Sung, and Bongjin Jun. Canny text detector: Fast and robust scene text localization algorithm. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.

[3] Homepages.inf.ed.ac.uk. (2019). *Image Analysis - Connected Components Labeling*. [online] Available at: https://homepages.inf.ed.ac.uk/rbf/HIPR2/label.htm [Accessed 4 Mar. 2019].

[4] M. Donoser and H. Bischof. Efficient maximally stable extremal region (MSER) tracking. In Proc. CVPR 2006, pages 553–560, 2006.