Word Hunter

Yang Zhao, Shuo Liu, Lingren Zhang
Department of Electrical Engineering, Stanford University, CA 94305

***Abstract*—For this project we propose and implement a reading tool on Andriod platform that can be used to identify keywords on paper-based media. We breakdown the image processing after receiving an image of the media into three steps. In the first step, we pre-process the image using binarization, de-skew and de-noising. The second step involves using OCR (in particular, Tesseract OCR engine) to recognize the text and find the keywords. Finally, we highlight the keywords by circling it with bounding boxes.**

I. Introduction

Have you ever read a long paper-based article and find yourself unable to locate keywords? With the advancement of digital media, sometimes we take basic word search for granted. But the world is still full of media printed on paper and it would make our lives much simpler if we can automate this basic reading tool for good old fashioned books. We propose a mobile application that will be able to find a word that the user specified through a phone's viewfinder. The phone highlights keywords detected in the viewfinder, saving the user a lot time manually searching for word.

For example, we want to search for "right" in this paper-based Declaration of Independence. We only need to use our smart phone to scan over the paper. Whenever the word "right" appears on the phone screen, it will be immediately circled in red bounding boxes.

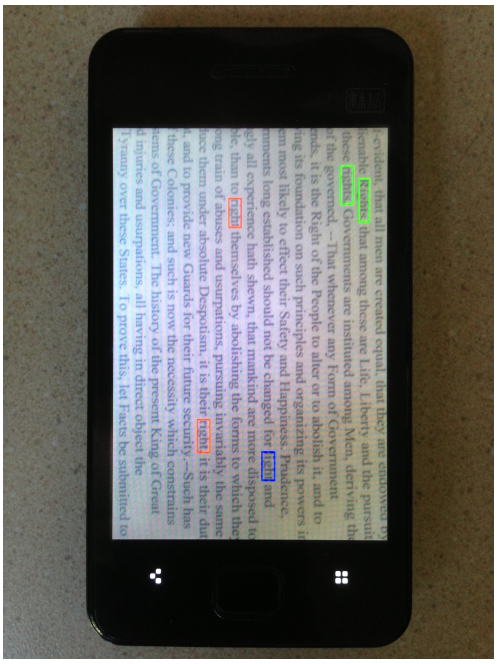

Fig. 1. Word Hunter Demo

II. Preprocessing

A. Binarization

The first step is to binarize the image in order to separate text from background in a grayscale image (which can be derived from RGB). Two methods were evaluated to be successfully applied in this project. The first method is Maximally Stable Extremal regions (MSER).[?] In computer vision, MSER is widely used as a method of blob detection. Like the SIFT detector, the MSER algorithm extracts from an image a number of co-variant regions, called MSERs. An MSER is a stable connected component of some level sets of the image. In this case, black words on the white paper are successfully detected by using vl-feat vl_mser function(MinDiversity=0.2, MaxVariation=0.7, Delta=15, MaxArea=0.1). The second method is locally thresholding. The reason we choose locally adaptive thresholding instead of global thresholding is that the lighting/brightness of the image is not uniform, which will cause global thresholding to perform poorly in the extreme bright/dark regions.

The idea of locally adaptive thresholding is divide the image into blocks/windows. For each block, use grayscale variance to determine whether the block is uniform. If the block is non-uniform (high variance), apply Otsu's method/global thresholding to the block; if the block is uniform (low variance), classify the entire block as all black or all white based on the mean grayscale value. The reason not to apply Otsu's method to every block is that if some blocks are background with a number of noise pixels, Ostu's method will keep the noise pixels while classifying the entire block as background will eliminate noise.

OpenCV function adaptiveThreshold is applied for binarization. It is observed that a blockSize of 51 yields a good trade-off between efficiency and thresholding effect. See Figure 2 and Figure 3 for an example of image before and after binarization.

B. De-skew

The input image from the user can be taken at an angle and the lines of text may not be horizontally aligned.[?] This may cause the following OCR stage to not perform as expected. So the image should be rectified first. Hough transform is used to detect (text) lines in the image. Then the binarized image is rotated by the mean of the rotation angles calculated from each text line. OpenCV function getRotationMatrix2D can be used to do it. In order to avoid cutting off corners in the rotation

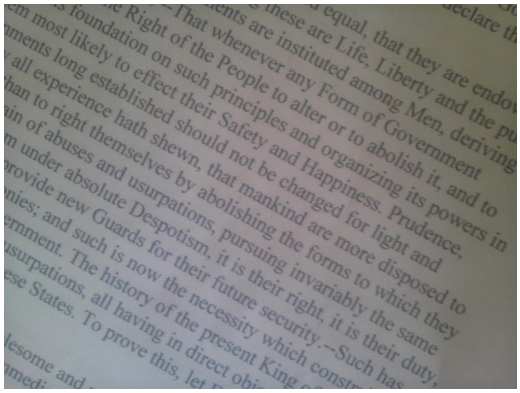

Fig. 2. Before Binarization

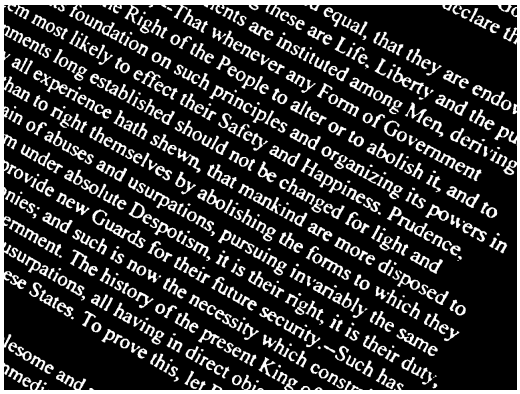

Fig. 3. After Binarization

process, padding is introduced before rotating. See Figure 4 for an example of image before and after de-skew. One can see that Figure 4 is larger than the images above because of padding, but the padding will be chopped off as a last step to match the original image size.

Observing that Tesseract does a decent job for images skewed by less than 5 degrees, we have included logic to bypass de-skew image rotation for such images. This improves accuracy because image rotation lowers image quality after interpolation. By skipping rotation, we essentially preserve more details in the image and therefore boost performance for images with a small skew.

We can see evidence of improvement in Figure 11, where 0-degree test cases have better performance than others. Even though theoretically 0-degree test cases should not be affected by image rotation if the skew is exactly 0 degree, we performed testing by hand-holding the device, which can introduce skew of a few degrees. Therefore skipping the rotation step can save us from the interpolation effect and boost performance.

C. De-noising

We also performed median filtering with a kernel size of 3 to eliminate any salt-and-pepper noise. This improves the accuracy of the character size detection, because otherwise

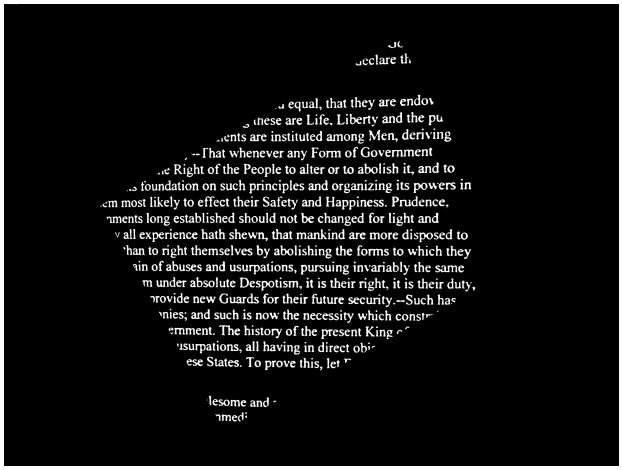

Fig. 4. Binarized And Deskewed

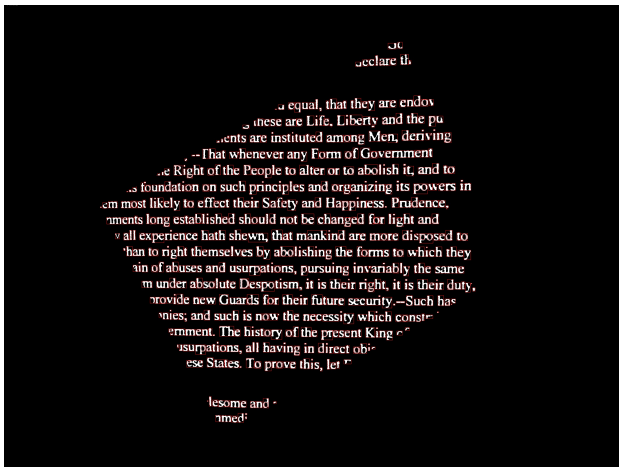

Fig. 5. Segmentation By Letter

the size of noise spots will introduce error into character size calculation.

III. Word Recognition

A. Segmentation By Letter

The first step of word recognition is to segment the image into rectangles each containing a letter. Because most letters consists of one connected component in the image, contours can be drawn (using OpenCV function findContours and drawContours) around each letter and then find a bounding box of each contour (OpenCV function boundingRect). See Figure 5 for an example of each letter surrounded by a rectangular bounding box.

Some letters (for instance, lower case “i”, “j”) consists of two connected components, and hence will have two bounding boxes, but it doesn’t affect the algorithm since they will get combined in the next step as we combine letter bounding boxes into word bounding boxes.

B. Smart Segmentation By Word

The motivation/assumption behind this method is that in a normal text document, linespacing, character size, word spacing, etc. scales linearly, i.e. large characters means large spacing, and vice versa. A similar idea is used in one of the previous projects.[?] We have to combine neighbouring letter bounding boxes into a word bounding box, but there is no existing OpenCV functions that performs this task. Therefore the following functions are implemented in C++:

1) Find Character Size: We implemented function findCharSize to estimate the average height and width of each letter (or its bounding box). This is important because when deciding whether two letter bounding boxes are neighbours, we need to look at the distance between them relative to the average size of bounding boxes. Making the decision based on absolute distance only is not accurate.

The implementation was done by creating a map with keys being the areas of the bounding boxes, and the values being the occurrences (i.e. how many bounding boxes has area equal to the key). Then the ten most frequent area values are selected, and their mean is used as the character size metric.

The function calculates the values for a few important parameters including:

- cArea: The mean calculated from the ten most frequent area values.
- cHeight: An estimate of average height of letters, calculated from $\sqrt{\text{cArea}}$ mutiplied by a constant factor to take into account that most letters have larger height than width.
- cWidth: An estimate of average width of letters, calculated from $\sqrt{\text{cArea}}$ multiplied by a constant factor to take into account that most letters have larger width than height.

2) Find Neighbour: Function isNeighbour is implemented to determine whether two letters are next to each other in the same word. This is the key logic in determining which letter bounding boxes to merge together into a word bounding box. Two bounding boxes need to satisfy both conditions below in order to be called neighbours:

- The x-coordinate of the right edge of box 1 and the x-coordinate of the left edge of box 2 are off by at most $0.32 \times$ cWidth (box 1 is to the left of box 2); or vice versa, the x-coordinate of the left edge of box 1 and the x-coordinate of the right edge of box 2 are off by at most $0.32 \times$ cWidth (box 1 is to the right of box 2). The factor 0.45 is the parameter that gives the best results after several experiments.
- Because different letters have different heights, we decided to use the y-coordinate of the bottom edge to determine whether two boxes are on the same row. The y-coordinates of the bottom edges of the two boxes needs to be within $0.32 \times$ cHeight of each other.
- In addition, we also want to combine the dot in lower case "i" and "j", therefore we allow the case where the y-coordinates of the top edges are within $0.28 \times$ cHeight.

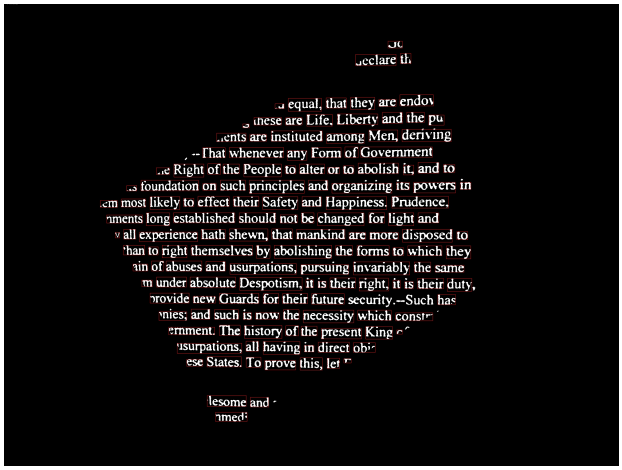

Fig. 6. Segmentation By word

The constant factors are tuned so that the algorithm works for the majority of test cases.

See Figure 6 for an example of segmentation by word.

C. Search And Label Matches

At last bounding boxes (hopefully each containing exactly one word at this point) can be passed to the Tesseract OCR engine. In order to improve efficiency, function withinRange is implemented to filter out boxes too wide or too narrow given the keyword length. Sometimes the image is out of focus or blurry due to vibration, therefore the result from Tesseract is not accurate. To cope with the imperfection, we label exact matches with red rectangles, non-exact matches with blue or green rectangles.

Exact or non-exact matches is determined by the ratio between edit distance (or Levenshtein distance) and the length of the keyword. A ratio of exact zero (or edit distance equal to zero) means that there is an exact match. A ratio reasonably close to zero means that we have a non-exact match.

Levenshtein distance is defined to be the minimum number of single-letter operations (insertion, deletion or substitution) needed to transform one string into another. For example, to change "abcd" into "bcda", one can either change each letter (change the first letter "a" to "b", the second letter "b" to "c", the third letter "c" to "d", the fourth letter "d" to "a"), which has a total of four operations, or delete the "a" from the beginning of the string and add an "a" to the end of it, which has a total of two operations. Therefore the Levenshtein distance between the two strings is two. The distance can be calculated between any two strings using dynamic programming.

Finally we draw rectangular boxes of different colors (in order to differentiate between exact matches and close matches) at the coordinates where we find matches, and then overlay the rectangles onto the orginal image. See Figure 7 and Figure 8 for illustration of searching and labeling keyword matches.

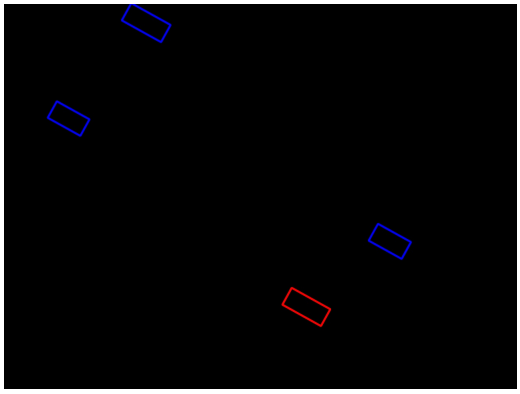

Fig. 7. Highlight Boxes

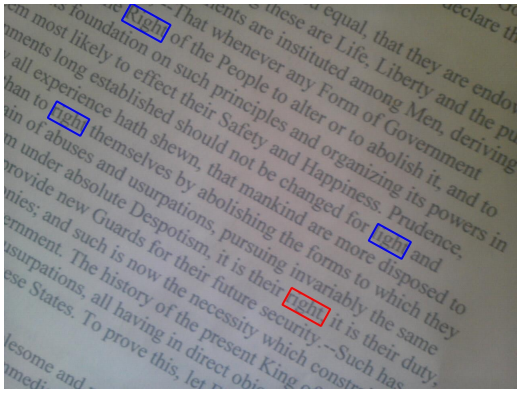

Fig. 8. Result Displayed

IV. Server Client Connection

OCR is an extremely computationally intensive operation. To perform this operation using the processors on an outdated smartphone would be impractical. Therefore, we decided to offload OCR and all other image processing operations onto the Stanford servers. The server side consists of two layers. The outer layer is a php script that waits for an http connection. Once the connection is established, the php script receives the image uploaded by the phone along with other parameters and stores them on the afs disk. However the server on which the php script lives does not have the necessary libraries to run OCR and OpenCV. Therefore, we implemented a backend python script that is responsible for linking up with the php script and executing the OpenCV/OCR executable on the Stanford corn server. Once the processing is complete, the php script picks up the output image and streams it back to the phone. Figure 9 illustrates the connection from client side/mobile device to Stanford servers to corn servers.

On the client side, WordHunter has three modes. The first mode, snap mode, takes a picture of the text when the user taps the screen. It then sends the image to the server for processing. After the image is received from the server, the phone will display it on the screen until the user hits the back button to enter a new word. User could also clear the screen by hitting the clear button to take another picture. The second mode,

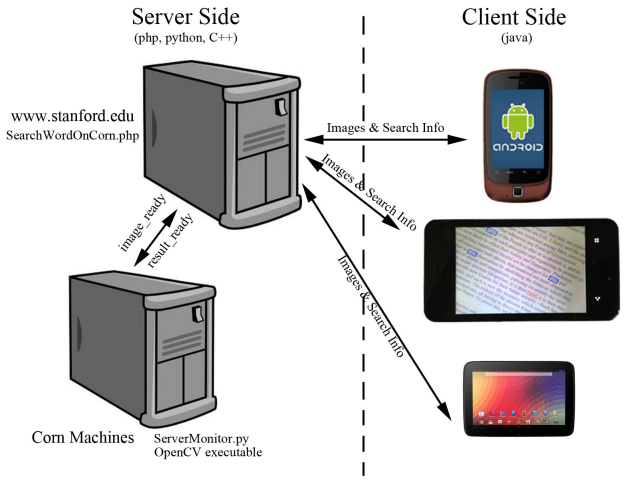

Fig. 9. Server-Client Connection

scan mode, continuously cycles through the above mentioned process seamlessly without any user intervention. Instead of displaying the image in the screen however, it will overlay a bounding box on top of the viewfinder to mark the desired word in real time. The third mode, hunter game mode, is a tentatively implemented game extension of the snap mode. This game is played between two players: First player could take a snapshot, then leave only the matched bounding boxes (without the original image) as a "clue" to the "secret treasure" for the second player. Then second player uses this "clue" to scan over the paper to find the perfect match. When the perfect match is found, the "secret treasure" set by the first user will be sent to the screen after the confirm button is hit.

V. Emperical Results

We tested the keyword recognition rate with more than 100 data points. The sample space covers different skew angles (0 degree, 15 degrees and 30 degrees), and different word lengths, because we believe that those are the two main factors that can affect performance. The overall accuracy is 89.7%, and we have included a breakdown by word length and skew angle (Figure10 and Figure 11).

From the breakdown by keyword length, we can observe that although accuracy varies for different word lengths, there is no clear trend that performance deteriorates as keyword length increases/decreases, which is desirable. Variation does exist but it is largely due to the relatively small sample size. We would like to perform more testing if time permits.

From the breakdown by skew angle, notice that there is a slight deterioration in performance due to the skew. We have investigated this by looking at the intermediate steps/results on the algorithm and concluded that the de-skew algorithm works reasonably well, the deterioration is mostly due to the fact that image rotation lowers the image quality. We also noticed that there is very little change in performance from 15-degree skew to 30-degree skew, which is desirable.

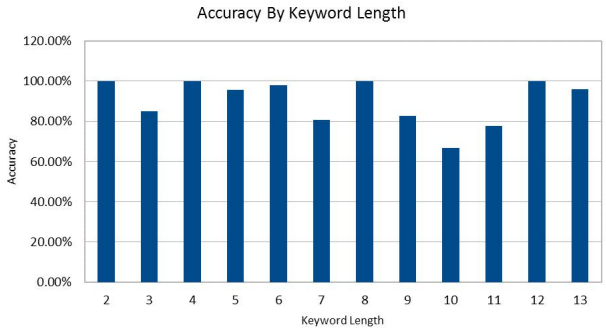

Fig. 10. Accuracy By Keyword Length

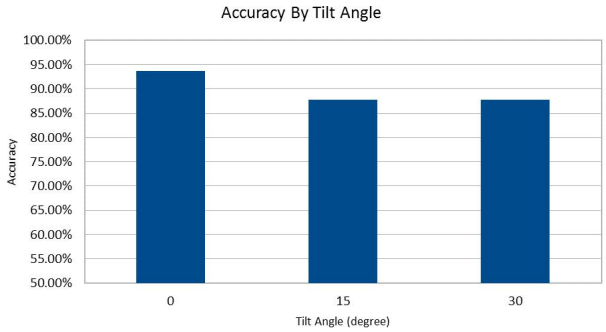

Fig. 11. Accuracy By Skew Angle

By examining the test input and output images, we noticed that results may vary due to uneven line spacing, different fonts, and image quality. De-focus and blur due to hand shaking account for a large proportion of inaccuracies.

VI. Conclusion and Future Work

We tried a variety of methods and techniques to successfully identify a word from a corpus of words within an image. With local thresholding, deskewing and smart segmentation, we were able to successfully segment each word in a majority of test cases. Both preprocessing and postprocessing (edit-distance based match analysis) were tailored to the powerful Tesseract OCR. Multi-platform (client, server) and multi-language(java, C++, python, php) were introduced in this project to bypass numerous configuration difficulties. For the purpose of demonstration, we were able to accurately query any word using an Android phone in the snap mode.

Ideally, our algorithm runs fast enough to support real-time word search. But due to the client-server transport delay, real-time scan-mode is impractical to use. To solve this problem, we could either move the computation to the client side, or use other feature detection (i.e. SURF) technique to reuse the previous detection results. Since consecutive frames can have large overlapping areas, we can increase efficiency by avoid re-computing those overlapping areas. Also, to improve user experience and make our app interesting, we could further integrate the voice-aided search and a more complex plot in our hunter game mode.

Acknowledgment

We would like to thank David Chen and Sam Tsai for their generous and responsive help on the project. We would also like to thank Professor Girod for his inspiring lectures.

Appendix

The team worked together on the image processing algorithm. In addition, Yang prototyped the Andriod app and server code, Shuo worked on the app as well as integration, Lingren worked on reporting.

References

[1] J. Matas, O. Chum, M. Urban, T. Pajdla, *Robust Wide Baseline Stereo from Maximally Stable Extremal Regions*

[2] Sam S. Tsai, Huizhong Chen, David Chen, Ramakrishna Vedantham, Radek Grzeszczuk and Bernd Girod, *Mobile Visual Search Using Image and Text Features*

[3] Vinay Raj Hampapur, Tahrina Rumu, Umit Yoruk, *Keyword Guided Word Spotting In Printed Text*