1. INTRODUCTION

For my final project, I decided to use computer vision techniques to recognize a check. The goal of my project is to recognize a photo of a check taken by a smartphone camera. I expect the check to be placed on a table, rotated in any direction. The program that implements this goal will find the check in the image, rotate it, and crop it, and recognize the text. Rather than attempting to recognize all possible text on the check, the scope is limited to the MICR (Magnetic Ink Character Recognition) codes at the bottom and the address at the top of the check.

Even though this project only analyzes photos, I envision this process as a step toward using real-time video from a smartphone camera to find a check and extract its data. There are many QR code recognition apps that will automatically find QR codes from the smartphone live video and will show you results without requiring the user to do much other than turn on the camera. I would eventually like to do the same for a check.

2. BACKGROUND

When proposing our final project ideas, each person in class selected two journal articles relating to our projects and presented a summary of one of the articles to the rest of the class. I selected Top-down and bottom-up cases for scene text recognition by Mishra [1] and Detecting and Reading Text in Natural Scenes by Chen and Yuille [2]. These articles deal with finding and recognizing text in natural scenes.

I presented a summary of the Chen and Yuille article to the class. It describes a process of finding text in photos taken in city environments. They based their work on the Viola-Jones face recognizer [3] which uses sub-window search of varying sizes, Haar features, and an AdaBoost cascading classifier to find faces in images. Chen and Yuille's text recognizer does similar analysis by using sub-window search, Haar features, and AdaBoost as well. However, they created specific Haar features specifically designed to find text and they also created very specific tests for recognizing text in the cascading Adaboost classifier. Although these texture-based algorithms were very interesting to learn about, I found them somewhat vague for the purposes of trying to reimplement their ideas myself.

The discussion that I found the most helpful for my project was the lecture on line fitting algorithms. In particular, the Hough Transform was very applicable to my project. The Hough Transform finds lines (or any parametric function like circles or ellipses) in an image by taking each point in an image and calculating every possible line that the point could be part of. As each point is selected, the points as a whole "vote" for lines. Once the algorithm is complete, you get a distribution of votes for all lines in the image.

The lines are represented in polar form, so the Hough Transform generates a table of rhos, thetas, and votes. Polar lines are used in the algorithm in order to avoid the problem of infinite slope when vertical lines are represented in rectangular form with slopes and intercepts. Since lines in polar form are represented as a line perpendicular to a ray of a particular magnitude (rho) and angle (theta), a vertical line can be represented with a given rho value and a theta value of 0 or 180 degrees.

3. METHODOLOGY

My check recognition method is broken down into four main steps: check segmentation from the image, rotation and cropping, text segmentation, and text recognition. Each of these is further broken into sub-steps. Interspersed between steps are tests to determine if a crucial step was successful. If not, it fails the process. Since I am attempting to recognize checks in photos with arbitrary rotations of the check, it is foreseeable that the process could fail. The tests are there so that recognition can fail as gracefully as possible.

3.1. Segment Check From Image

Segmenting the check from the image is the most important step because it highly influences whether or not text recognition will be successful. This step determines what the boundaries of the segmented check are. If the boundary is incorrect, then it will impact the effectiveness of subsequent steps.
3.1.1. Hough Transform

The main purpose of using the Hough Transform is to find the lines representing the edges of the check. In order to find the edges, I first apply the Sobel filter to the image. This will create a black and white image in which strong edges of the original image become white pixels on a black background.

Once I have a black and white image, I run it through the Hough Transform checking thetas from -90 to 89 degrees with steps of 1 theta. I tried smaller steps of theta, but I did not get better results compared to the cost of computational time. Once I had the resulting matrices from the Hough Transform, I used custom code rather than built-in Matlab functions, to find the edges of the check. I tried the built-in functions in Matlab like houghpeeks or houghlines, but I was not able to get the results that I wanted. I found it more effective to analyze the Hough Transform data myself.

3.1.2. Line Selection

The main assumption I make when analyzing the Hough Transform data is that the line with the highest response will correspond to a line parallel with the long edges of the check. Since I am not sure if that response will be an edge of a check or a strong line in the middle of the check, I only use the theta value from that response. I use that theta to calculate a perpendicular theta, and calculate min and max ranges for the two thetas using an "angle tolerance" value of 5 degrees. If my highest Hough response had a theta of 62 degrees, I would calculate the perpendicular angle with either theta - 90 (if theta is positive) or theta + 90 (if theta is negative). For example, if the highest response were 62, the perpendicular would be 62 - 90 = -28, and I would create ranges like [57, 67] and [-33, -23].

My process for selecting lines from the highest Hough responses developed with much trial and error. Even though I could filter the lines using ranges of thetas, there were still thousands of possible lines. At first, I used a relatively simple calculation that worked for some of my test checks. I sorted the Hough table by number of votes, selected the top 100 lines, sorted by rho magnitude and then selected the first and last rows. By selecting the first and last rows, I was assuming that I would get the top and bottom or left and right edges.

Selecting the lines by magnitude of rho worked fine for many of my check sample images. However, it became a problem for noisier images in which maybe my target lines were not in the top 100 voted lines. I realized that the top 100 was an arbitrary selection because there are over 2 million possible lines in the Hough Transform dataset. So in order to select the highest voted lines, I used the standard deviation and max to find a number of results to keep. Because the algorithm slows down as the number of rows increase, I capped the size at 1,000.

I then found that sometimes I was selecting lines that were very close to each other. In order to solve that, I changed my algorithm to enforce a minimum specify a minimum difference of rho that each parallel line must be from the other. To implement that, I take a table of rhos, thetas, and votes and partition it into 2 tables that have rhos a minimum distance from the max rho value in the table. I set the minimum difference to be 10% of the largest possible rho value.

3.1.3. Image Crop

Once I find 4 lines, I have a 4x2 matrix of (Rho, Theta) pairs representing lines in polar form. I then take those and convert them to rectangular form with the formula

\[ r = x \cos(\theta) + y \sin(\theta) \]

In order to turn them into lines, I take a set of x values (the number of columns in the image) and use them to find y with the formula: \[ y = (r - x \cos(\theta))/ (\sin(\theta)) \]. With an array of points, I use polyfit to convert those points into lines that can be used with Matlab's fzero function to find intersections of lines to find the 4 corners of the check. Once I calculate the coordinates of the corners, I then calculate a bounding box and use imcrop to crop the check out of the larger image.
3.2. Image Rotation

Once the check is found and cropped, my next goal is rotating the check to a horizontal orientation. One nice side effect of using the Hough Transform is that it gives me angles of each line in terms of the angle of rho. In order to find the angle of the line itself (rather than the angle of rho), I calculate the perpendicular angle theta to find the angle of my line. Since the photo may have been taken at an angle not directly above the check, there may be some perspective skewing in the image. So rather than only using one of the angles of the long edges, I take an average of the angles of the long edges.

3.2.1. Add Padding Before Rotation

At this point, I know the right angle to rotate, but I want to control how the check rotates so that pixels are not lost at an edge. In particular, I want the point of rotation to be about the center of the check. I find the center point by finding the intersection of diagonal lines from the corners of the check. One technical issue I encountered is that imrotate will rotate an image around the center of the image, but I could not find parameters to force it to rotate around a particular point. So in order to make the rotation work, I add padding to the image so that the center of the check is at the image center. Another issue I found is that a rotation may cut off part of the image. So, I further pad the image so that the height and width are the size of the diagonal of the image's previous size. That will allow it to rotate the full 360 degrees without any pixels disappearing off of an edge. Once the second padded image is created, then I can successfully rotate the image.

3.2.2. Rotate Corner Points

After rotating the image, the next goal is to crop the rotated image. However, since I rotated, my corner points no longer match up with the check. I handle this by rotating the points. Throughout the process of cropping and padding the check image, I also made adjustments to the locations of the corner points in order to correctly track the pixels of the check on the screen. So to keep the corner points lined up with the check, I rotate the points about the center of the image using the same angle of rotation as the check image. To calculate this, I multiply a rotation matrix by a vector representing my point to get the location of the new point: \[ \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix} \]

One the corner points are rotated, I calculate a bounding box for those points in order to crop the image again. If the prior steps were reasonably successful, the check should be closely cropped and the lines of text should be nearly horizontal in the image.

3.3. Segment Text from Check

The next goal is segmentation of the text. Since some checks have backgrounds with lively pictures (I have checks with beach scenes), I try to remove the background by basing my segmentation off of the Sobel filter. I then use bwlabel to find connected components in the filtered image, filtering by area to try to reduce the noise (of tiny points being selected). In order to try to keep my segmentation size invariant, I experimented with a few area sizes and then I calculated an area factor by dividing a given area I selected by the total area of the image. That way, if the size of the image changes, I am looking for a size similar to the relative size I am targeting.

3.3.1. Segment MICR Text

I begin segmenting with the bottom of the check. Since all checks use a standard MICR (Magnetic Ink Character Recognition) font for routing number, account number, and check number, it simplifies the task of recognizing those characters. The fonts have very distinctive shapes, are consistent in height and letter spacing. I base my segmentation mostly off of the consistent height of the fonts. In order to find the exact set of bounding boxes to select, I use a horizontal line to scan vertically and count how many bounding boxes intersect the line. I find the peak number and select the middle y position of the line and select boxes by testing which boxes intersect with the line.

Once I have the bounding boxes for the MICR data, I segment by using the boxes to crop from the grayscale image (not the bw Sobel image). I then use Bradley adaptive binarization on those grayscale images. The adaptive binarization helped to reduce the effects of shadows which had resulted in some digits having some parts disappear with standard im2bw.

3.3.2. Segment Address Text

I segment the address text in a very simple manner. Assuming that the check is correctly cropped in a horizontal orientation, I crop the check image by half the width and one third the height from the top. I select the region from the grayscale of the image and then run Bradley adaptive binarization to generate a black and white image. I had originally used the built-in binarization, but some parts of the text would disappear.

3.4. Text Recognition

3.4.1. Recognize MICR Text

I decided to perform my own ocr on the MICR font in a similar way I had done in project 3. Since the MICR font is so unique, Matlab's built-in ocr function does not work at all. I decided to recognize the text using simple Euclidean distance to classify the characters. In order to perform classification, I needed to create a training set. I
created the training set by first finding a free font library for the MICR font. Then I used Mac OS's font book, grab, and Photoshop to create a set of separate font images. I loaded the files into Matlab and applied erosion, dilation, and 5 rotations in order to create a set of 15 sample images per digit. I also cropped and resized those images to try to match what I did with the ones found on the check.

In order to recognize the digits, I used a KNN classifier that I created for project 3. With a similarly sized training set, I got the best results when I used k=1. Since only one k value was chosen, the classification was purely based on finding the match that had the lowest Euclidean in the training set.

3.4.1. Recognize Address Text

Using the cropped black and white image that I took from the top of the check image, I then use Matlab's OCR function to recognize the text. That function adds in extra line spacing, so I filter that out before displaying the results.

4. RESULTS

In order to test my algorithm I took photos a 7 different checks that I took with my iPhone 5S with resolutions of 3264 × 2448. Some checks are from closed accounts and some are checks that credit card companies send in their marketing campaigns. I took photos of the checks rotated in various ways, including near horizontal and near vertical orientations. I took the photos on two different tables, one that has a dark red color and the other has a lighter wood color. The lighter wood has all kinds of spots and lines, which adds some noise to the image around the check. I included those to see how well the algorithm would work with a noisy background.

Of the 41 total checks, 35 of the 41 (85.3%) were found by the algorithm and processed to recognize the text. 6 of 41 failed because a check the lines that were chosen in step 3.1.2 (line selection) did not pass a test to make sure the check has a target aspect ratio. I test for the aspect ratio indirectly by finding the angle of intersection between the diagonal lines. By measuring one of the checks I had, I observed that the acute angle should be around 48 degrees and the obtuse angle should be about 132 degrees. I test for that angle with a tolerance of 10 degrees.

Of the checks that had been processed, the number of checks that had all everything correct (address, routing number, account number, and check number) was 10 of 35 (28.5%). The number of checks that had all of the MICR data correctly recognized was 28 of 35 (80%). The MICR text recognition was likely the highest because that font is so unique that even a simple Euclidean distance classifier could correctly identify each character. The ones that were not correctly identified were typically due to poorly cropped checks and certain images that had the word "VOID" appearing in the Sobel filtered images. As long as every character could be correctly segmented, then they would be correctly recognized.

On the other hand, I got very inconsistent results with Matlab's built-in OCR function. In order to differentiate between correct and incorrect results, I was only looking if the correct string appeared in the results at all. It the text was correct but had some extra garbage before or after, I still counted that as correct. More advanced semantic analysis of the text results should be able to filter most of that out. Aside from that, the number of exactly correct address results were 13 of 35 (37.10 %), results with 1 to 2 errors were 11 of 35 (31.42%), and 3 or more errors were 11 of 35 (31.42%). More precise segmentation would likely improve those results, and I would imagine a higher quality OCR engine would improve it as well.

5. CONCLUSION

I found this to be a very challenging project and overall I am happy with the progress I made. I know that the algorithm could continue to be improved, but I think that I generally went in the right direction. I continue to think of ways to improve my algorithm, especially the process of segmenting the check from the photo.

If I were to continue to work on this, I might experiment with trying to fit a rectangle of the proper aspect ratio to the lines found from the Hough transform. Also, since the built-in OCR results were so poor, I might try to implement the texture analysis I saw in the articles I read in order to localize the text. With more sophisticated segmentation of the text, better OCR should be possible.

11. REFERENCES