CP467 Report

Daniel Berezovski - 160173040
Amardeep Sarang - 160112080
Kumiko Randle - 160660010

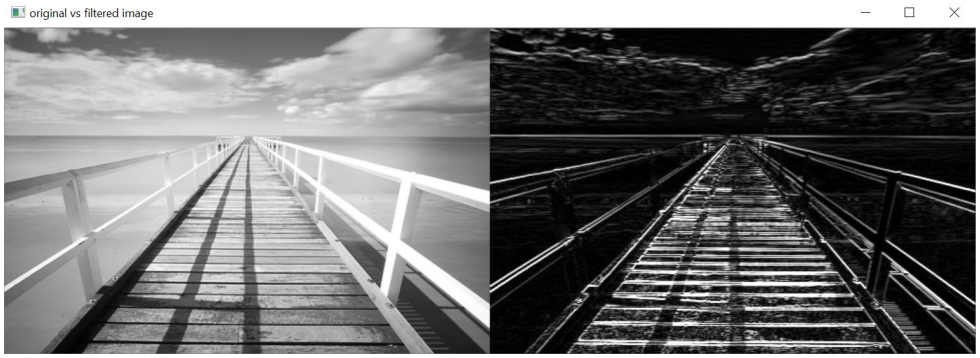
Abstract

This paper will discuss several image processing techniques and utilities that can be used in the implementation of optical character recognition (OCR) software. These utilities include convolution filters, scaling, and connected region recognition. Some of these utilities were used to create an OCR that used K-nearest neighbors algorithm (k-nn) and 10000 handwritten digits from the MNIST digit dataset as training data. The effects of different k value and not using a thinning algorithm are also discussed.

Utilities

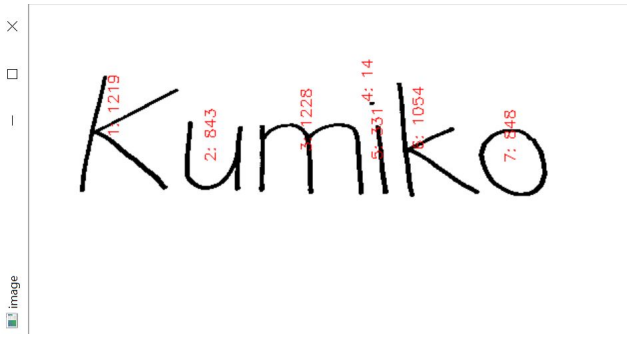
A1

This program implements image filtering, a foundational concept of in computer vision which allows for the extraction of low-level features from images. In this implementation a user is able to specify both an input image and their choice of filter, and then they are shown the resulting image juxtaposed with the original. Seen below are the results of a vertical edge detection filter (top) and a horizontal edge detection filter (bottom) applied to an image of a pier. This is achieved by taking the gradient over the image's pixel intensities along either the x or y axes. For instance, consider the filter [-1/2, 0, 1/2] - when applied to an image patch this is the same as taking the (discrete) derivative along the x-axis (as defined by the first principal, discrete $f'(x) = (f(x+1) - f(x-1))/2$). In our 3x3 filter we perform the same operation (adjusted by a constant factor), however since we take it over 3 rows instead of just 1, the filter becomes more robust and less sensitive to noise.



A2

This program detects connected regions of dark pixels in an efficient manner. This is done by first applying a threshold over all pixels in the image, setting them to be either completely black or completely white. Then we map each pixel to a region ID - all pixels which share the same ID belong to the same region. We can do this by iterating across pixels in the image over the scan line (left to right, top to bottom), and if a of the neighbouring pixel which has been previously visited has a region ID, we set the current pixel to have the same ID. In the case that multiple regions neighbouring the current pixel, we merge the regions together, and give them all the same ID. Finally we number the regions and print the number of pixels in the region. This is displayed at the mean location of all the pixels in the region, as seen below.

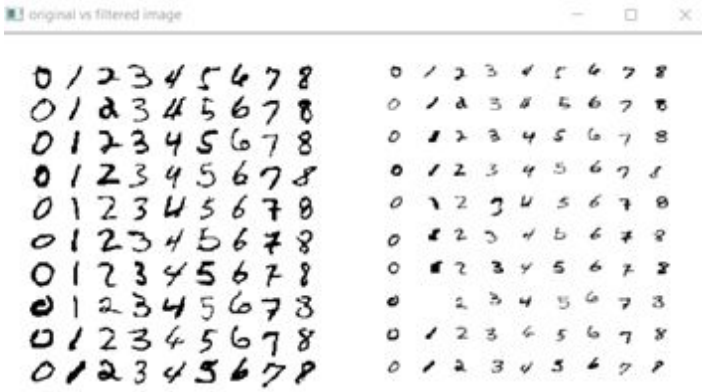


A3

In this assignment we used a combination of the 2 different methods to achieve the same result, demonstrating the separability of convolution matrices, and how it can prove to be useful in a real world setting. To accomplish this task, we applied a 3x3 mean filter over an image, showing its smoothing effects, and contrasting it from the original in a noticeable fashion. The 3x3 mean filter is also considered a separable filter, allowing us to demonstrate the phenomenon of producing the same effect with multiple filters, which add up to become a separable filter. In the case of the 3x3 mean filter, we are able to create its same effect on an image by using a row filter of $[\frac{1}{3}\ \frac{1}{3}\ \frac{1}{3}]$, and a column filter of the same values. Applying each of these filters individually produces the same results as applying a 3x3 mean filter, and the output of this operation can be seen below.

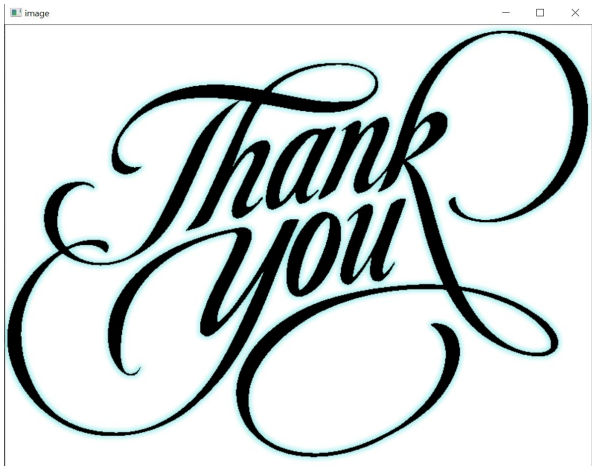
A4

This program scales down all symbols in the image to use specified dimensions and then places them back in the image at their original locations. The program uses findContours function from the OpenCV python library to build a bounding box around each symbol. It then uses another OpenCV function resize, to resize the bounding box to the user-specified dimensions and in turn scales the symbol. A sample of the output of this program is shown, in this example, the symbols were scaled to 10x10 pixels. This program is used in the OCR to ensure all symbols are the same size and intern the feature vector is also the same size.



A8

In this assignment we combine a variety of previously implemented image processing techniques to create a new and text-augmentation tool, called the “glow” text effect. The input image should ideally be a greyscale image of text, although other line-based images will yield equally good results. We then apply a threshold over the pixel intensities, setting them to either completely black or completely white. Next an edge detection image filter is used to pick out the edges of the lines in the image. We then use a large mean filter and then additional thresholding to thicken these lines (thresholding any pixel > 0 to be 0). Next we blur these thickened lines and apply a color to them, before finally overlaying the original image on top. In doing so, we add a visually appealing glowing effect to the image. This new image operation has many artistic usages in the real world, such as to create word art for the title of a document.



Optical Character Recognition

A5

This program divides any given image into 9 zones, keeping each zone to the same size if possible. It goes without saying that each zone is a square, divvying up the image into a 3x3 grid. We then set a threshold value of 127 (the exact midpoint between 0 and 256, the boundaries of pixel brightness), and used this threshold to classify each pixel as white or black. After doing this for each zone, we calculated the ratio of black to white pixels, and used this number to represent the zone in a feature vector. Each feature vector will have 9 elements, 1 for each row. The results of this process can be seen below, under **A6**, where digits 0-9 are presented with their feature vectors.

A6

0: [0.01010101 0.60714286 0.01123596 0.30434783 0.39655172 0.39655172 0.09756098 0.6875
0.02531646]
1: [0. 0.11111111 0. 0. 0.30645161 0. 0. 0.15714286 0.]
2: [0.07526882 0.5 0. 0.05882353 0.5 0. 0.07142857 0.42105263 0.24615385]
3: [0.2345679 0.23287671 0. 0.11111111 0.84090909 0.28571429 0. 0.35 0.24615385]
4: [0.01010101 0.08433735 0.07142857 0.23287671 0.44642857 0.09459459 0. 0.19117647 0.
]
5: [0.03092784 0.26760563 0.26760563 0.42857143 0.5 0.0125 0. 0.47272727
0.02531646]
6: [0.05263158 0.15384615 0. 0.25 0.58823529 0.35 0.07142857 0.35 0.0125]
7: [0.07526882 0.25 0.05882353 0. 0.20895522 0.08 0. 0.35 0.]
8: [0.01010101 0.30434783 0.16883117 0.15384615 0.92857143 0.22727273 0.15384615 0.24615385 0.
]
9: [0.01010101 0.21621622 0. 0.08433735 1.38235294 0.02531646 0. 0.0125
0.17391304]

OCR Procedure

Our OCR is able to recognize multiple written digits from the same image. To do this we first would change the image to black and white. Then we used our connected region utility function to detect each digit and get the corner coordinates of its bounding box. Using the bounding boxes we could scale each digit down to the same size. From here we could create comparable feature vectors for each digit using the black and white ratios in the 3x3 zones. These vectors were then normalized using min-max normalization formula x_new= (x_old-min)/(max-min), where min and max are the minimum and maximum values in the vector X. We could then use training data and k-nn to classify each digit.

Machine learning algorithm used

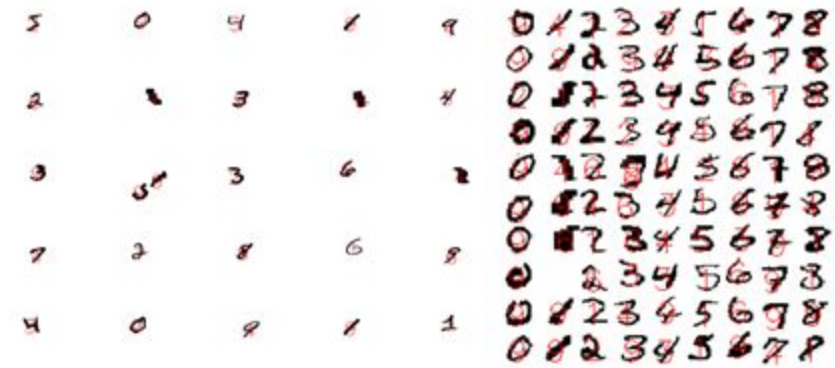
We decided to use the K-nearest neighbors algorithm (k-nn) for classifying digits. K-nearest neighbor is a similarity-based algorithm. It calculates the Euclidean distance from the target input to each available training vector and makes a classification decision based on the k nearest training vector using a majority voting policy. In it's simplest form, k=1, k-nn acts simply as a Euclidean distance measuring algorithm were it matches the target vector to the closest vector training vector. With larger values of K, it can be generalized, and become more rebuts to the varying examples of the same class, for example were different styles of writing the same digits. It is also easy to train k-nn on different training datasets since it does not require pre-training and can be trained at runtime. These considerations all contributed to us using k-nn in our OCR.

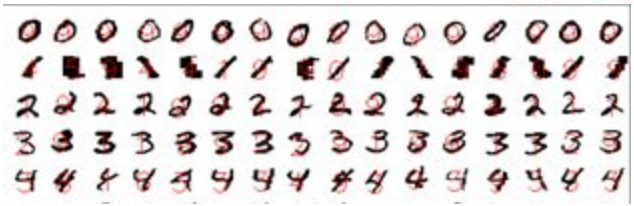
Data set used for training

To train the K-nn we used the data from the MNIST handwritten digit dataset. From the MNIST data set, we took 100 sample images for each digit and we used the methods explained above to generate a feature vector for each sample image and stored them in a file. Using this method, we generated a training data set consisting of 1000 feature vectors that would be used to train k-nn. We choose to have a training set of 1000 (100 of each digit) because we predicted that it would help k-nn generalize the classes and allow it to adapt to different styles of writing the digits, while also keeping performance at a reasonable level.

Results

Once k-nn had classified the digits, the program would display the original black and white image and then overlay the predicted classification for each digit in red, as can be seen, it the test samples images seen below. In these samples we tested using the a K value of 3 and we scaled all symbols to 20x20 pixels.





As can be seen, not all of the symbols are handled perfectly by our methods. The program is unable to handle the "5" in test 1 which was not originally continuous, it treated it as two distinct symbols and gave each part a separate classification. It should also be noted that the scaling algorithm has not worked perfectly in all cases. One of it's most notable failures is how it deals with the "1" digit in all tests where it thickens them and can even be seen turning them into black blobs.

Due to these and other shortcomings, the accuracy rate for these tests was very low. The accuracy rate for tests 1, 2, and 3 were calculated to be 4%, 11%, and 4% respectively.

Despite seeing low accuracy across the board, we did observe some small improvements with larger K values where the average accuracy for K=9 saw a small improvement from that of K=3. These results can be seen in the table below.

K value	Test 1	Test 2	Test 3	Test 4	Average
1	8%	6%	4%	6%	6%
3	4%	11%	4%	6%	6%
9	12%	4%	13%	0%	7%

Evaluation

In the aftermath of these low accuracy results, we have noticed several patterns that can explain them. One of the patterns that we noticed was the fact that digits that took up more space in there bounding box, had a long horizontal line, and were written more thickly generally lead to them being classified as a 9, 8, or 6. Whereas letters that were think and tightly written lead to them being classified as a "1". We can also observe that in many circles and half-circles such as those in 9s and 5s have been largely ignored which could be due to the fact that the circles were thickened too much by the scaling algorithm.

Conclusions and further research

From this, we can come to the conclusion that despite using a fairly conventional feature set and a straight forward classification algorithm and a large amount of training data our OCR suffered from a lack of a thinning algorithm that would have thinned the digits. This problem was made worse by the fact that the digits were often made to look thicker by the scaling algorithm. It is highly recommended that any future attempts made to improve this OCR make use of a thinning algorithm. It may also be fruitful to explore the use of alternative scaling algorithms and to explore what affect different scaling factors have on accuracy. It is also recommended that the effect of the K-nn K value be explored in more depth in future studies.