FACULTY OF ENGINEERING AND APPLIED SCIENCE

SOFE 2715U DATA STRUCTURES FINAL PROJECT REPORT

Data Structures Final Project
Section CRN: 73375

IMAGE BARCODE GENERATOR AND IMAGE SEARCHING

A. INTRODUCTION

This report proposes the use of image searching and generating barcodes to annotate images. There are many possible ways to go about how to create a barcode to present each image with a high accuracy for an example, some of the ways are adding more than 4 projections which would then increase the retrieval accuracy by a few. Other than that, we could also utilize the majority voting of n retrieved images, we can use weighted voting on distance and we can also use the optimal number and corresponding optimal angles of the projection. In addition, another way would be instead of using the threshold strategy we can convert the real integer values directly to the binary numbers because thresholding causes loss of information. Moreover, another method would be creating the binary numbers based on other strategies or even based on the threshold value less than 0 or bigger and finally we can forget about using the binary barcode and instead keep the real value/integer and then use the equilibrium distance to retrieve a better accuracy. For this particular project though, we were given a database of ten images per each set of handwritten digits, 0 to 9, which means our database contained 100 images in total. Using the images from the database we had to generate barcodes corresponding to each individual image. We then compare each barcode of each digital image to then find the most similar case of a given query in the dataset, with respect to accuracy. Furthermore, to retrieve a high accuracy percentage we decided to use the method which required us to increase the number of projections greater than 4. Nonetheless, Content-Based Image Retrieval (CBIR) as of today is one of the most popular ways to retrieve query images by using images in the archive to then detect similarity by colour, shape, texture, etc. Moreover, the basic necessity to retrieve and browse images from the dataset or archive is to search and sort the most similar image with minimal human interaction. Likewise, since most people are acquainted with using the internet in their daily lives and as a result adhire data from the web it has become fundamental for quick search and retrieval engines. Therefore, this paper tries to go more in depth about how effective an image barcode generator and image search/retrieval engine is and how it works. As well as, using projection angles to minimize the chance of error, hence contributing to a generate more effective barcodes, which would then help to retrieve the most similar image and a more impactful accuracy.

B. ALGORITHMS

First of all, what is an algorithm?

A sequence of instructions called an algorithm is used to execute a given operation. This may be a straightforward operation like multiplying two numbers or a more complicated operation like playing a compressed video file. For particular requests, search engines use proprietary algorithms to show the most important results from their search database.

This section covers the algorithms for our code. The final project had two main algorithms, that is, barcodeGenerator algorithm and imageSearch Algorithm.

Algorithm 1: barcodeGenerator()

Description: This algorithm scans all the images in our dataset named "MNIST_DS" and creates a barcode of all the images depending on our **method** of creating barcodes. Furthermore, it stores all the barcodes in a 10X10 2-d array in which the column of that 2-d array represents the image itself (img_0, img_1, img_2, img_3, and so on) and the row represents the subfolder (0, 1, 2, 3 and so on). The functions p1, p2, p3, p4 are basically the projection functions at different angles. p1 represents 0 degree, p2 represents 45 degree, p3 represents 90 degree and p4 represents 135 degree. thresholdCalculator returns the threshold array depending on the projection which is being passed and the size of that projection.

```
Step 1: Start
Step 2: Assign our image database to a variable named directory.
directory ←'MNIST_DS'
Step 3: Run a loop through the directory in order to access the elements of the directory.
table ←[] #initializing the 2d table array
for subdir in directory:
        col ← [] #initializing the columns for the 2d table array
        for file in subdir:
Step 4: Assign the image which is being assessed to a variable  img
           img ← PIL.Image.open(file)
Step 5: get the image data
           image_sequence ← img.getdata()
```

```
Step 6: Store the image sequence in a 1d array using numpy
        imgArray ← np.Array(image_sequence)
Step 7: convert that 1d array into a 2d array using numpy
        imgArray2d ← np.reshape(imgArray, (28,28))
Step 8: Store the four different projection in four different variables
        projection1 ← p1()
        Projection2 ← p2()
        projection3 ← p3()
        Projection4 ← p4()
Step 9: Calculate the threshold frequency for each projection and store it in
different four different variables
            Th_p1 v thresholdCalculator(projection1, len(projection1))
            Th_p2 ← thresholdCalculator(projection2, len(projection2))
            Th_p3 ← thresholdCalculator(projection3, len(projection3))
            Th_p4 ← thresholdCalculator(projection4, len(projection4))
Step 10: Concatenate the 4 threshold projection arrays from each projection
using numpy and store it in a variable
            RBC ← np.concatenate(Th_p1, Th_p2, Th_p3, Th_p4)
Step 11: Add the RBC (Radon barcode) to each column of the table array for
each row (since we're in a loop) and then add all the columns to the row of the table
array
           col.append(RBC)
          table.append(col)
Step 12: Return the table array in the end.
return table
```

Algorithm 2: imageSearch(barCodeArray)

Before we move on to the description of this algorithm it is important to **understand what is hamming distance?**

The Hamming distance is a metric that can be used to compare two binary data strings. Hamming distance is the number of bit locations in which the two bits vary by comparing two binary strings of equal length. If d reflects the Hamming distance between two strings, a and b, then it is denoted as d(a,b).

Lets understand the concept of hamming distance with an example:

Example 1

$010 \oplus 011 = 001$, $d(010, 011) = 1$.

Example 2

$010 \oplus 101 = 111$, $d(010, 101) = 3$.

Example 3

$010 \oplus 111 = 101$, $d(010, 111) = 2$.

Description: This algorithm searches for the image depending on the hamming distance between the barcode of the image being searched and the image being compared. The combination with the lowest hamming distance is considered as a match for the image being searched.

This algorithm further calculates the hit ratio which is basically the success rate of our method. If the image being returned with the lowest hamming distance is of the same subfolder as the image being searched then the hit value is incremented by 1 otherwise not. In the end, the search algorithm prints the image which is being searched and the image which is being compared and returns the hit ratio.

Step 1: Start

Step 2: initialize some variables which would be further used in our function

```
ham ← 0
imgClass ← 0
hit ← 0
img1Row ← 0
img2Row ← 0
img1Col ← 0
img2Col ← 0
```

Step 3: Run a nested for loop which would assign the variable imgSearch a barcode from the 2d array which we had returned in the barcodeGenerator algorithm. imgSearch is the barcode of the image which is being searched.

```
for i in range(0, len(barCodeArray)):
    for j in range(0, len(barCodeArray[i])):
        imgSearch ← barCodeArray[i][j]
        img1Row ← i
        imgCol ← j
        shortest ← len(imgSearch)
```

Step 4: Run another nested for loop inside our previous nested for loop which would assign the variable imgCompare different barcodes from the 2d array which was returned from barCodeGenerator algorithm.

```
for x in range(0, len(barCodeArray)):
    for y in range(0, len(barCodeArray[x])):
        imgCompare ← barCodeArray[x][y]
```

Step 5: Calculate the hamming distance between the imgSearch and imgCompare using Scipy library.

```
ham ←distance.hamming(imgSearch,imgCompare)
```

Step 6: compute the lowest hamming distance

```
if(ham!=0 and ham<shortest):
    shortest ← ham
    img2Row ← x
    img2Col ← y
    imgClass ←x
else:
    pass
```

Step 7: Print the image with the which is being searched and the image being compared. There will be 100 search results.

```
imgOne ← plt.imread('path of the directory of the image being search' )
imgTwo ← plt.imread('path of the directory of the image being compared')
plt.imshow(imgOne)
plt.imshow(imgTwo)
plt.show()
```

Step 8: Calculate the hit ratio

```
If (i==imgClass):
    hit+←1
```

Step 9: Return the success rate/hit ratio

```
return hit
```

C. REQUIRED MEASUREMENT AND ANALYSIS

When approaching the problem of designing, implementing and testing a content-based image retrieval system, it was obvious that hit ratio, and complexity would be some of the utmost important factors. Hit ratio is undeniably the most important factor of the software as it determines how often the algorithms are correct. With the following in mind, the hit ratio within the search_algorithm was found to be roughly 50%.

When determining the Big-O complexity analysis for both the Barcode_Generator and SearchAlgorithm they were found to be very closely the same. This can be noticed as the Barcode_Generator algorithm has a Big-O complexity of *O(N^2)*, whilst the SearchAlgorithm has a larger complexity of *O(N^4)*. Both of these Big-O complexities are very large but can be justified by the results that they return.

Calculations for Big O Analysis

Barcode Generator

Set database to variable directory ←'MNIST_DS'	**Operations** 1

Loop through all of the data
table ←[] #initializing the 2d table array
for subdir in directory:
col ← [] #initializing the columns for the 2d table array
for file in subdir:
	1
	N
	N
	$\frac{N(N-1)}{2}$

Open the image img ← PIL.Image.open(file)	$\frac{N(N-1)}{2}$
Assign the image data image_sequence ← img.getdata()	$\frac{N(N-1)}{2}$
Get the image data imgArray ← np.Array(image_sequence)	$\frac{N(N-1)}{2}$
Convert 1-d array to 2-d imgArray2d ← np.reshape(imgArray, (28,28))	$\frac{N(N-1)}{2}$

Create a each projection and calculate threshold frequency
projection1 ← p1()
projection2 ← p2()
projection3 ← p3()
projection4 ← p4()
Th_p1 v thresholdCalculator(projection1, len(projection1))
Th_p2 ← thresholdCalculator(projection2, len(projection2))
	$\frac{N(N-1)}{2}$
	$\frac{N(N-1)}{2}$
	$\frac{N(N-1)}{2}$
	$\frac{N(N-1)}{2}$
	$\frac{N(N-1)}{2}$
	$\frac{N(N-1)}{2}$

Th_p3 ← thresholdCalculator(projection3, len(projection3))
Th_p4 ← thresholdCalculator(projection4, len(projection4))
	$\frac{N(N-1)}{2}$
	$\frac{N(N-1)}{2}$

Concatenate the projections together RBC ← np.concatenate(Th_p1, Th_p2, Th_p3, Th_p4)	$\frac{N(N-1)}{2}$

Add the RBC to each column of the array
col.append(RBC)
table.append(col)
	$\frac{N(N-1)}{2}$
	N

Return the table return table	**1**
Total Operations:	$\frac{15*N(N-1)}{2}$ + 3*N + 3

Search Algorithm

Initializing Variables
ham ← 0
imgClass ← 0
hit ← 0
img1Row ← 0
img2Row ← 0
img1Col ← 0
img2Col ← 0
	Operations
	1
	1
	1
	1
	1
	1
	1

Nesting loops to assign barcodes
for i in range(0, len(barCodeArray)):
 for j in range(0, len(barCodeArray[i])):
    imgSearch ← barCodeArray[i][j]
    img1Row ← i
    imgCol ← j
	N
	$\frac{N(N-1)}{2}$
	$\frac{N(N-1)}{2}$
	$\frac{N(N-1)}{2}$

shortest ← len(imgSearch)	$\frac{N(N-1)}{2}$
	$\frac{N(N-1)}{2}$
Assign a barcode to each image	
For x in range(0,len(barCodeArray)):	$\frac{N(N-1)(N-2)}{6}$
For y in range(0,len(barCodeArray[x])):	$\frac{N*(N-1)*(N-2)*(N-3)}{24}$
imgCompare ← barCodeArray[x][y]	$\frac{N*(N-1)*(N-2)*(N-3)}{24}$
Calculate the hamming	
ham ←distance.hamming(imgSearch,imgCompare)	$\frac{N*(N-1)*(N-2)*(N-3)}{24}$
Compute the lowest hamming distance	
if(ham!=0 and ham<shortest):	$\frac{N*(N-1)*(N-2)*(N-3)}{24}$
shortest ← ham	$\frac{N*(N-1)*(N-2)*(N-3)}{24}$
img2Row ← x	$\frac{N*(N-1)*(N-2)*(N-3)}{24}$
img2Col ← y	$\frac{N*(N-1)*(N-2)*(N-3)}{24}$
imgClass ←x	$\frac{N*(N-1)*(N-2)*(N-3)}{24}$
else:	
pass	$\frac{N*(N-1)*(N-2)*(N-3)}{24}$
Output the images that are being matched	
imgOne ← plt.imread('path of the directory of the image being search')	$\frac{N(N-1)}{2}$
imgTwo ← plt.imread('path of the directory of the image being compared')	$\frac{N(N-1)}{2}$
plt.imshow(imgOne)	$\frac{N(N-1)}{2}$
plt.imshow(imgTwo)	$\frac{N(N-1)}{2}$
plt.show()	$\frac{N(N-1)}{2}$
Calculate the hit ratio	
If (i==imgClass):	$\frac{N(N-1)}{2}$
hit+←1	$\frac{N(N-1)}{2}$

Return the hit ratio Return hit	1
Total Operations:	$8+N+12*\frac{N(N-1)}{2}+\frac{N(N-1)(N-2)}{6}+9*\frac{N*(N-1)*(N-2)*(N-3)}{24}$

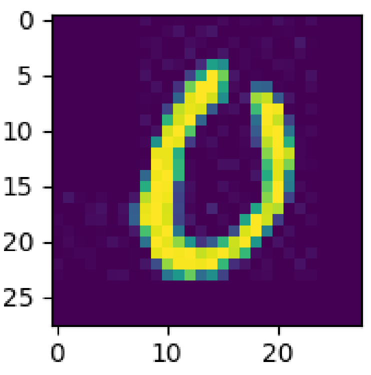
D. SEARCH RESULTS

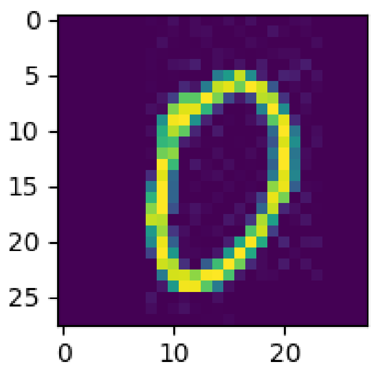
This section highlights the search results of our code. The snippet on the top of each class depicts a successful search result (in which the hit ratio is incremented) and the snippet on the bottom of each class depicts an unsuccessful search result (in which the hit ratio is not affected).

The search result is considered successful when the image being returned with the lowest hamming distance is from the same class as that of the image being searched.

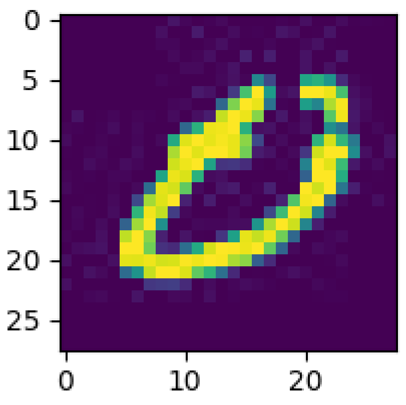
Image class 0

Successful





Unsuccessful



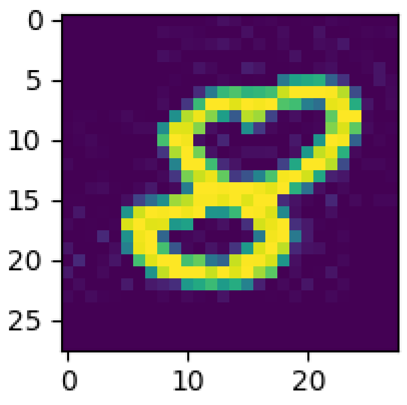
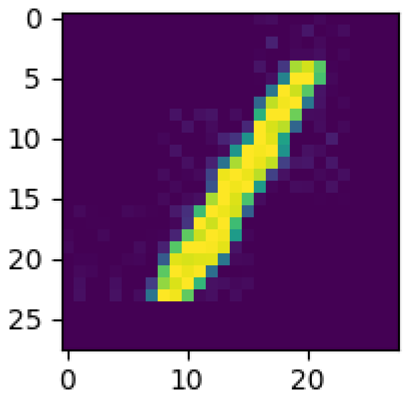
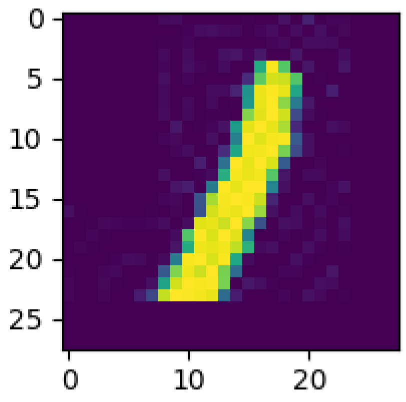


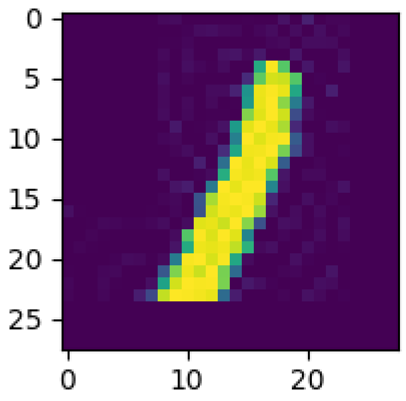
Image class 1

Successful





Unsuccessful



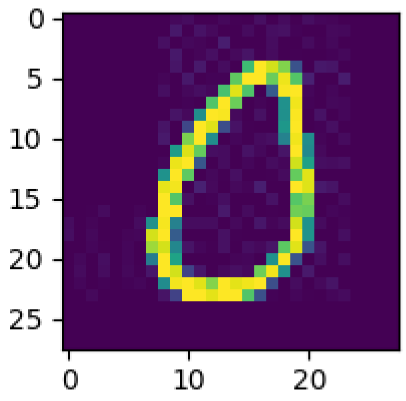
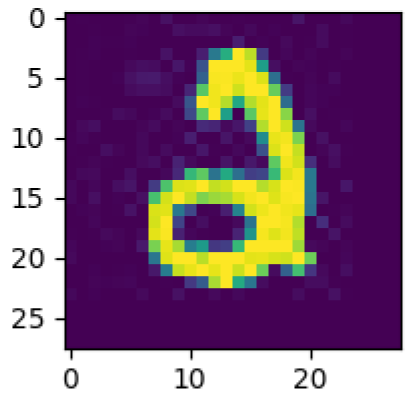
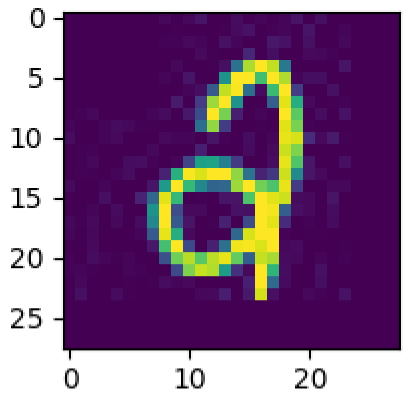


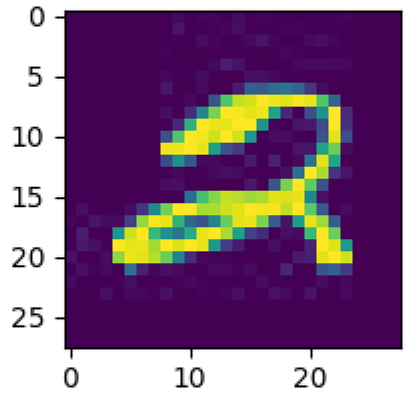
Image class 2

Successful





Unsuccessful



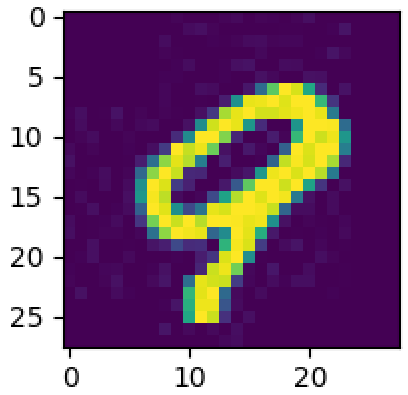
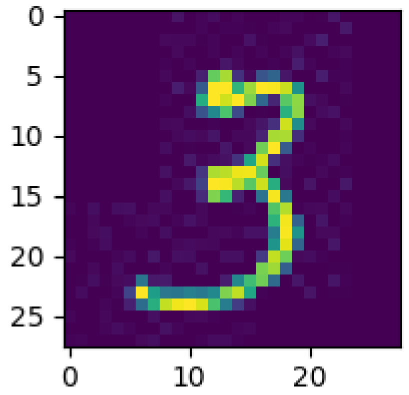
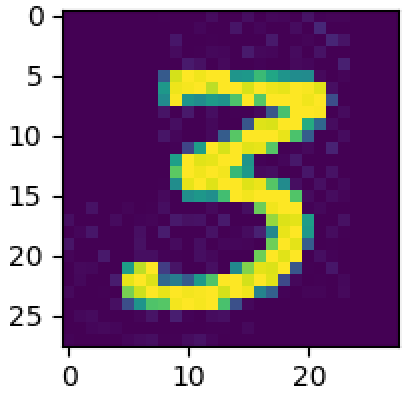


Image class 3

Successful





Unsuccessful

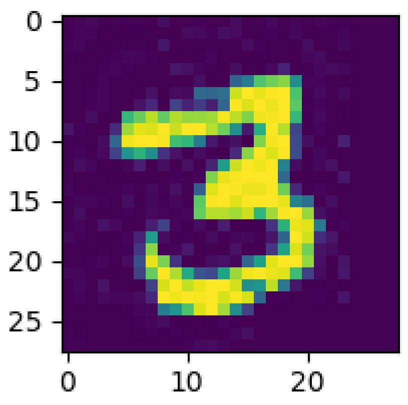
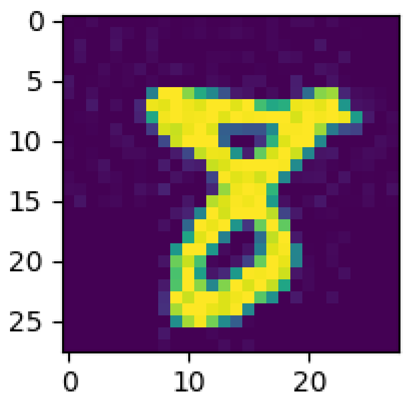
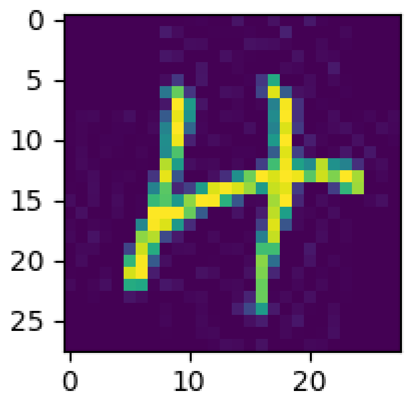
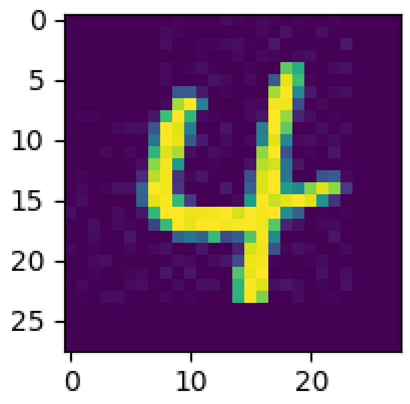



Image class 4

Successful

Unsuccessful

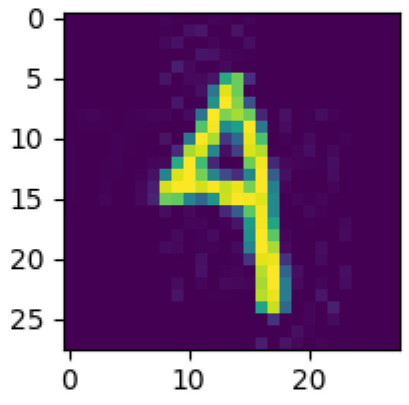
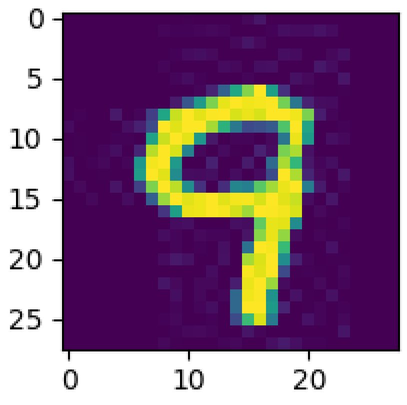
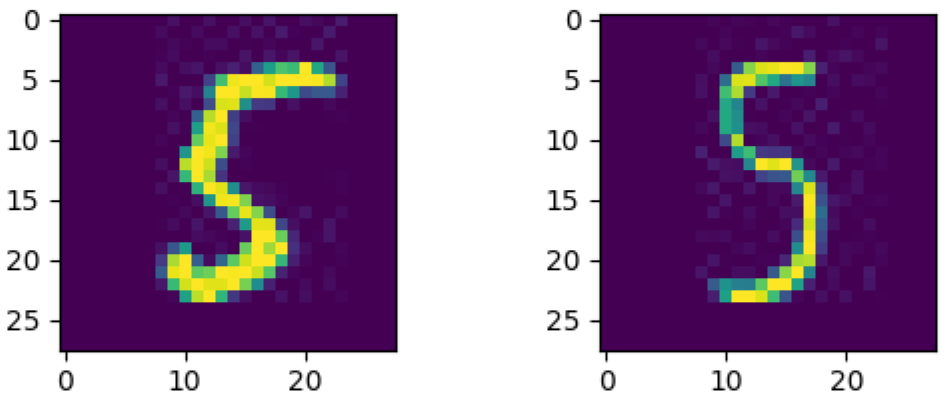



Image class 5

Successful



Unsuccessful

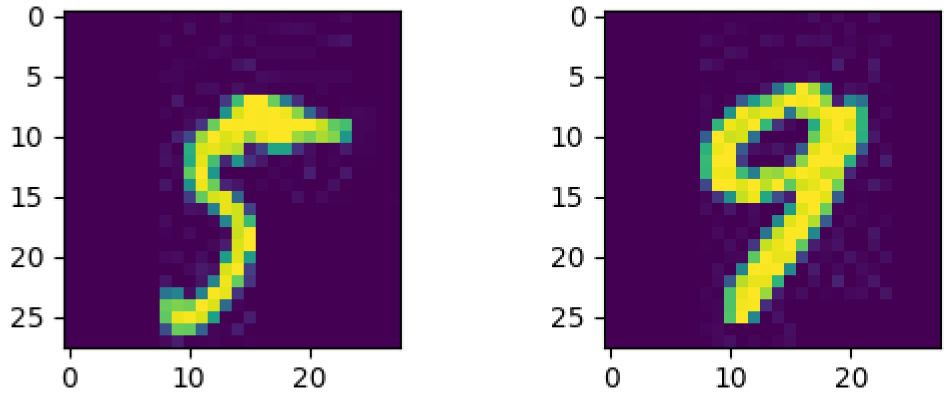
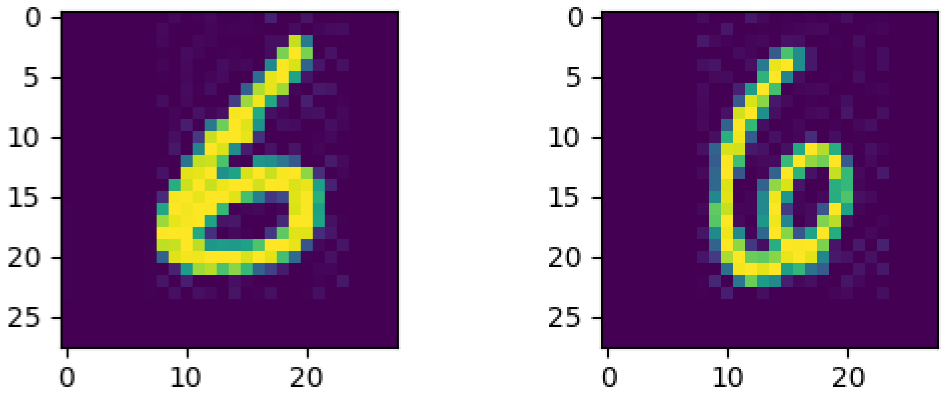


Image class 6

Successful



Unsuccessful

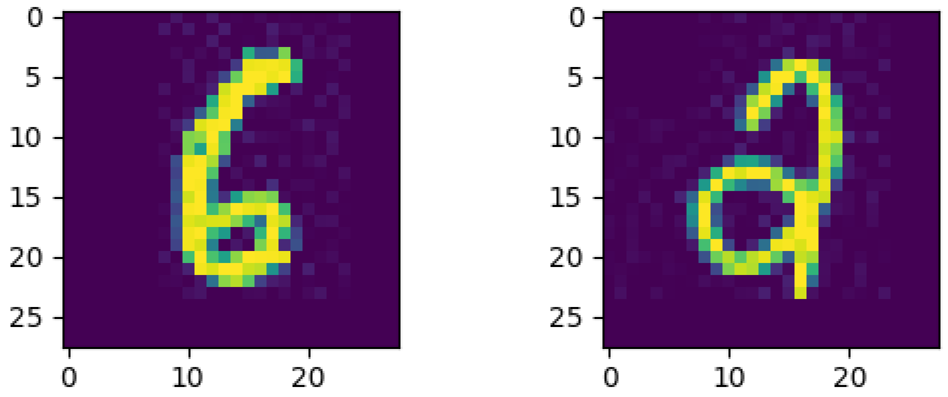
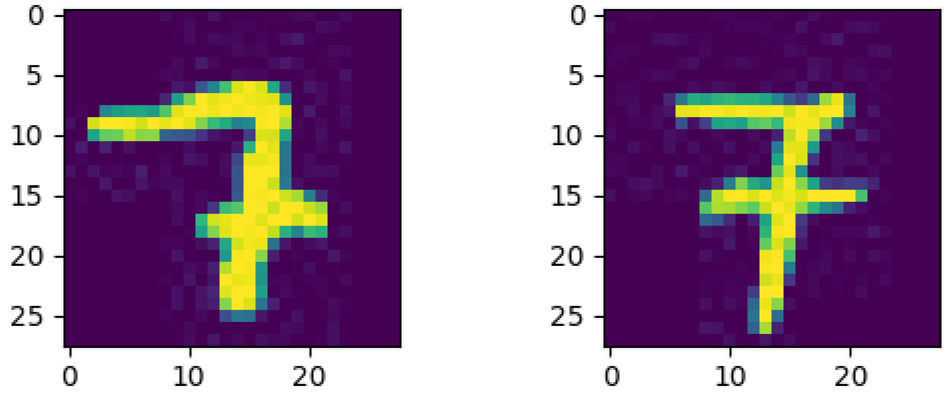


Image class 7

Successful



Unsuccessful

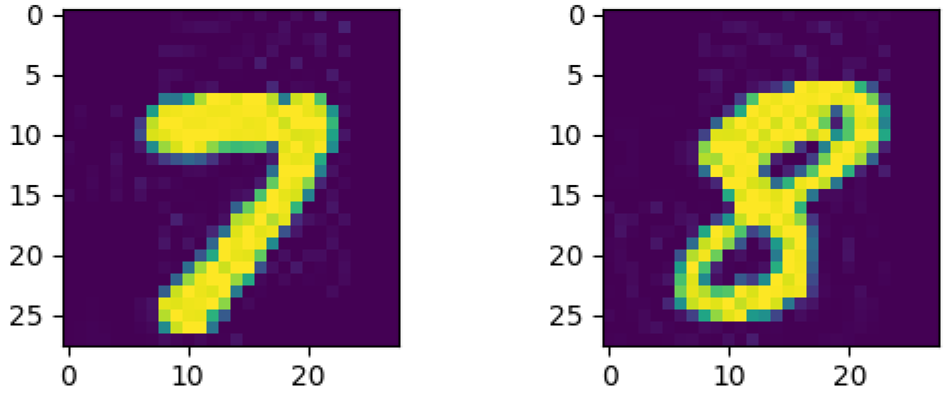


Image Class 8

Successful

“No successful snippet for class 8”

Unsuccessful

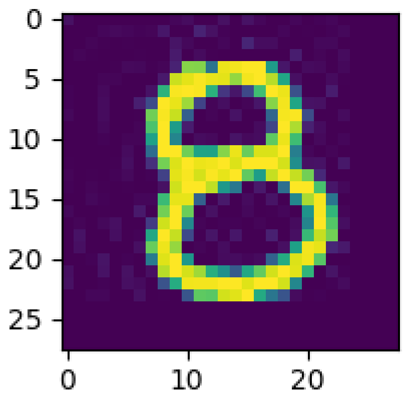
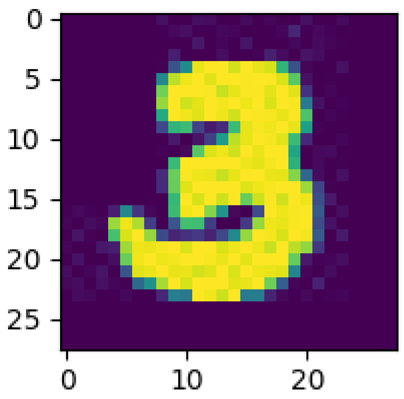
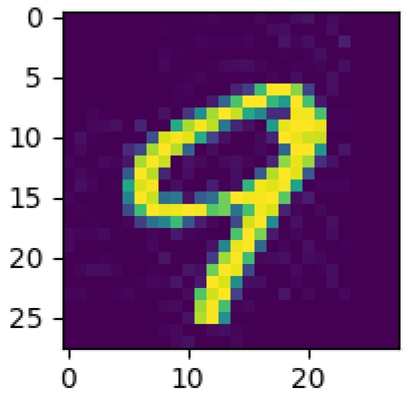
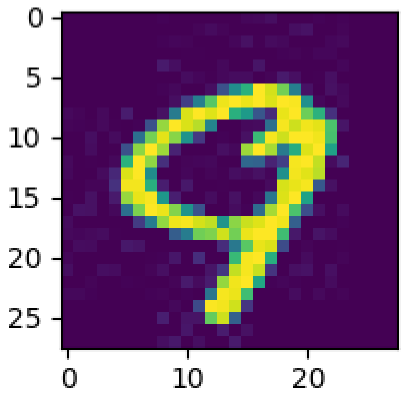
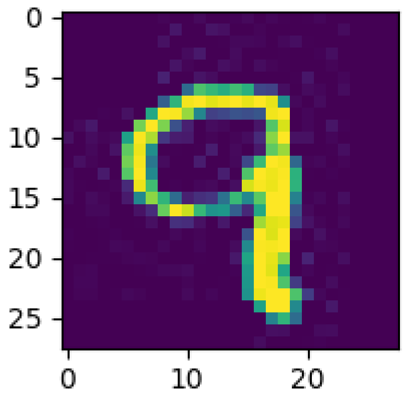
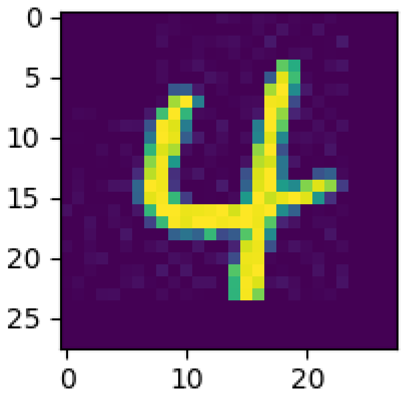



Image class 9

Successful

Unsuccessful

E. CONCLUSION

To conclude, this report provides how to solve a real-world problem such as barcode generating and retrieving similar images from a database of several sets of images. Throughout this project searching the images was not an issue. However, we did have some difficulty generating barcodes as when used projections and corresponded them to their respective angles. From that we then used each corresponding threshold value and assigned it 0 or 1 regards to if the value was equal or less than the threshold value. From the projection values we found ourselves struggling to figure out how to calculate the projection angle for 135 and 45 degrees and how to incorporate that into the loops as well as what condition to pass. As a result, hitting a success was a bit harder. Therefore as a result, we were only able to hit an accuracy ratio of about 50%. Overall, this project though we encountered difficulties we managed to write a workable code that generates barcodes corresponding to each image. This project has been a great learning experience and should mold us for related tasks in the future.