# Content Based Image Retrieval On Handwritten Digits Using Barcode

by

Daniel Gohara Kamel,
Jessica Leishman

SOFE 2715U: Data Structures
Dr. Shahryar Rahnamayan

Ontario Tech University
Oshawa, Ontario, Canada
April 11, 2021

## Introduction

This report seeks to explore the use of content based image retrieval through the use of handwritten digits and the radon barcode method. Using a text-based image search can only be as accurate as the annotations on an image; utilizing an image as a query yields new data possibilities and patterns not inherently visible to the human eye. This leads to new and unique ways to approach the typical keyword search.

Images are composed of pixels, which are a series of bits that describe the colour to be displayed. This program procedurally analyzes these images by taking various projection angles that emanate from the centre of the image, as this is an effective way to get several "snapshots" of what the image might be. This was then used to generate a barcode and compare this to other images to find similar images.

For this program, the images in the MNIST_DS folder provided are used as both the user_given_dataset and the dataset to compare against.

## Required Packages and Folders

The zip provided in the project submission should contain an empty folder called matches, which is typically required to be created by the user in order to use the program. If you do not see this, please create it.

In addition to this folder, please ensure the following packages are installed on your system:
opencv
numpy
Scikit-image
matplotlib
pillow

## Algorithms

### Barcode Algorithm:

**Description:** get_images() function crops and returns an array of all the images in the given dataset. create_code() creates the barcode for the query image. create_code_set() writes all created barcodes from the given dataset to a text file.

**Parameters:** create_code() has a parameter which is the query image.

```
Algorithm Barcode_Generator
function get_images()
    initialize cropped array
    for all image in given_dataset
            crop image to rows & column 4-22
            append image to cropped array
    return cropped

function create_code(image)
    Initialize code array

    for angle from 0 to 165 degrees incrementing by 15 degrees
        projection <- radon projection of angle
        threshold <- mean of projection array
        for i from 0 to length of projection
            if (projection[i] >= threshold)
                projection[i] <- 1
            else
                projection[i] <- 0
        append projection to code array
    code_string <- initialized to empty string
    for size of code array
        convert code array entries to string data type and append to codeString
    return codeString

function create_code_set()
    create 'codes.txt' file if does not exist or open if file does exist
    images = get_images()

    for each image in the dataset
        code <- create_code(image)
        write code to end of codes.txt file
    close file
```

## Explanation:

get_images() iterates through each image in the dataset folder and crops it to create 18x18 images. This array is appended to an empty three-dimensional array. The function returns this array of images, where indices 0 through 9 are zeroes, 10 through 19 are ones, 20 through 29 are twos and so on, so the quotient (integer) of the index divided by 10 would be the number that barcode represents.

create_code() takes radon projections of the image from 0 degrees to 165 degrees (inclusive), incrementing by 15 degrees each time. The projection is saved within a one-dimensional array. If each array entry from the projection is greater than or equal to the threshold value calculated (using the mean of the pixels in the projection), a 1 is placed in the barcode. Otherwise, a 0 is placed. Once each projection has been completed, this barcode is converted into a string datatype, and the string format is returned.

create_code_set() calls get_images() and create_code() to create the barcodes for the dataset. It then creates or opens a text file and saves the created barcodes in the file, called codes.txt. Each line corresponds to an image in the dataset, which allows it to be quickly called by its line. codes.txt will be overwritten each time the algorithm is run.

## Search Algorithm

**Algorithm** Search_Algorithm

**Description:** find_one() is responsible for comparing the query image with all the barcodes and finding the closest match to the query image's barcode. find_all() function is responsible for calling the find_one() function for all the images in the users_dataset in order to find a match for every image in their dataset.

**Parameters:** The parameter for the find_one() function is query_image a 2D, 28 by 28 array representing an image of a hand written number. The parameter for the find_all() function is the users_dataset, an array of 2D arrays, each representing a 28*28 image of a handwritten number.

```
function find_one(query_image)
    query_image <- query_image cropped to rows & columns 4-22
    code_string <- create_code(query_image)

    open and access text file with codes of images
    lines <- array of each barcode in text file
    close text file

    distance <- 216 (length of code_string+1)
    index_store <- None

    for k from 0 to length of lines
        line <- lines[k]
            if length of code_string is not equal to length of line
                Throw Exception
            else
```

```
                for j from 0 to length of code_string
                    if code_string[j] == line[j]
                        distance incremented by 1

                    if distance <distance_store and distance != 0
                        distance_store <- distance
                        index_store <- k

    print distance_store
    print code_string
    print lines[index_store]

    number_match <-  int(index_store/10)
    print number_match


function find_all(user_dataset)

    for image in user_dataset
        find_one(image)
```

## Explanation:

find_one() calls create_code() with the query image and stores the created barcode in the code_string variable. Then accesses the text file containing the created barcodes dataset. Hamming distance is then used to compare both barcodes to find level of similarity. Every barcode is compared against the query image to find the closest match. The query image and the closest match barcodes are then displayed with their hamming distance and the number contained in the match.

find_all() calls find_one() with each image in the users_dataset.

## Required Measurements and Analysis

### Big-O Complexity for Barcode Algorithm:

```
Algorithm Barcode_Generator
function get_images()
    initialize cropped array                                1
    for all image in given_dataset                        100
            crop image to rows & column 4-22              100
            append image to cropped array                 100
    return cropped                                          1
```

```
function create_code(image)
    Initialize code array                                                    1
    for angle from 0 to 165 degrees incrementing by 15 degrees              12
        projection <- radon projection of angle                             12
        threshold <- mean of projection array                               12
        for i from 0 to length of projection                      12*18 = 216
            if (projection[i] >= threshold)
                projection[i] <- 1
            else
                projection[i] <- 0                                 12*18 = 216
        append projection to code array
    code_string <- initialized to empty string                               1
    for size of code array                                                 216
        convert code array entries to string data type and append to codeString   216
    return codeString                                                        1

function create_code_set()
    create 'codes.txt' file if does not exist or open if file does exist     1
    images = get_images()                            O(1) from analysis = 1
    for each image in the dataset                                            n
        code <- create_code(image)             n*O(1) from analysis = n*1 = n
        write code to end of codes.txt file                          n*1 = n
    close file                                                               1
```

## Analysis:

create _code_set() operates the algorithm. create_code_set() creates the greatest time complexity of O($n^2$). It first calls get_images(). The greatest time complexity in get_images is O(1), as the loop runs 100 times (the size of the given database). The for loop then calls the create_code() function n times (for each image to be matched). create_code() has a time complexity O(1), because the lines in the program run a fixed number of times. Therefore, the Big O time complexity of the algorithm is O(1) * O(n) * O(1) = O(n).

## Big-O Complexity for Search Algorithm:

```
function find_one(query_image)
    query_image <- query_image cropped to rows & columns 4-22        1
    code_string <- create_code(query_image)                          1

    open and access text file with codes of images                   1
    lines <- array of each barcode in text file                       1
    close text file                                                   1
```

```
distance <- 216 (length of code_string+1)                          1
index_store <- None                                                1

for k from 0 to length of lines                                    100
    line <- lines[k]                                               1
        if length of code_string is not equal to length of line
            Throw Exception
        else
            for j from 0 to length of code_string          100*216=21600
                if code_string[j] == line[j]                       21600
                    distance incremented by 1                      21600

                if distance <distance_store and distance != 0      21600
                    distance_store <- distance                     21600

                    index_store <- k                               21600


print distance_store                                               1
print code_string                                                  1
print lines[index_store]                                           1

number_match <-  int(index_store/10)                               1
print number_match                                                 1


function find_all(user_dataset)
    for image in user_dataset                                      n
        find_one(image)                  n*O(1) from analysis = n*1 = n
```

## Analysis:

The code in the find_one() function has a time complexity of O(1). The for loop contained in this code runs from 0 to the number of barcodes in the text file, this would have a time complexity of O(1) as it runs for the size of the given dataset (100). The find_all() function contains a for loop that runs from 0 to the number of images in the user dataset to be matched against the given dataset, this has a time complexity of "n". This for loop then calls the find_one() function which contains a time complexity of O(n). Therefore, the total Big O time complexity of this algorithm is O(1) * O(n) = O(n).

## Retrieval Accuracy in terms of Hit Ratio(%):

The retrieval accuracy of the program created is 73%. Each of the 100 images within the dataset were used as the query.

Each category had varying success rates, as can be seen in the breakdown below:

**Table 1: Hit Ratio Breakdown of 12 Projection method:**

| Digit: | Correct Match Percentage by Digit | Digit of Incorrect Matches: |
|---|---|---|
| 0 | 90% | 1 |
| 1 | 100% | N/A |
| 2 | 90% | 8 |
| 3 | 70% | 4,5,6,7 |
| 4 | 70% | 6,6,6 |
| 5 | 50% | 3, 4, 6, 6, 9 |
| 6 | 80% | 4, 5 |
| 7 | 70% | 1, 2, 9 |
| 8 | 50% | 5, 7, 9, 9, 9 |
| 9 | 70% | 4, 7, 7 |

**Table 2: Barcode Generation and Search Algorithm Times of 4 and 12 Projections:**

| Method: | 4 Projections | 12 Projections |
|---|---|---|
| Hit Ratio<br>(cropped images) | 68% | 73% |
| Average* time to generate barcodes (ms) | 95.535755157470703125 | 231.7333221435546875 |
| Average* time to find query index 1 match (ms) | 4.9550533294677734375 | 13.066530227661132813 |
| Average* time to find query index 50 match (ms) | 6.9320201873779296875 | 18.4078216552734375 |
| Average* time to find query index 100 match (ms) | 5.9814453125 | 15.038728713989257813 |

| | | |
|---|---|---|
| Average* time to find single query match (ms) | 5.95617294311523 | 15.5043601989746 |
| Average* time to find and save single query match (ms) | 232.150872548421 | 225.95238685607910156 |
| Average* time to find all matches (ms) | 358.28065872192382813 (0.3583 seconds) | 673.38228225708007813 (0.6734 seconds) |
| Average* time to find and save all matches (ms) | 17575.805187225341797 (17.5758 seconds) | 18359.216213226318359 (18.3592 seconds) |

*each average time was taken from 5 sample runs of the program under the same conditions.

## Results Analysis

Process Analysis:
Searching using the original images un-cropped using the 4 recommended projections 0, 45, 90, 135 degrees resulted in a hit accuracy of 51%

Increasing the number of projections from the recommended 4 to 9 using 15 degree increments allowed the program to achieve a hit accuracy of 59%. These projections were taken from [0°,135°].

The images were then cropped to remove blank space that made a border around most of the digits in the dataset. They were first cropped to a 20x20 pixel image, using rows and columns 3 to 23 of the image-array, which yielded a hit accuracy of 67%. This was further refined to an 18x18 pixel image, using rows and columns 4 to 22 to produce a hit accuracy of 69%. This final crop of rows and columns 4 to 22 was found by trial and error, and was the best setting found for this data set.

Further increasing the projections to 12, using 15 degree increments over [0°,165°] resulted in an accuracy increase of 4%, increasing the final hit accuracy to 73%.

Running Time Analysis:
With 4 projections, the barcode generation takes 95.535755157470703125 ms (0.0955 seconds). The time to generate the matches for the entirety of the dataset is 358.28065872192382813 ms (0.3583 seconds), and the time to generate these matches and save them to the test computer is 17575.805187225341797 ms (17.5758 seconds).
The hit accuracy was 68%.

The time to generate a single match with 4 projections for dataset index 1, 50, and 100 were 4.9550533294677734375 ms (0.0050 seconds), 6.9320201873779296875 ms (0.0069 seconds), and 5.9814453125 (0.0060 seconds) respectively.

The time to generate a single match and save it to the test computer with 4 projections for dataset index 1, 50, and 100 were 211.60602569580078125 ms (0.2116 seconds), 216.46332740783691406 ms (0.2165 seconds), and 268.38326454162597656 ms (0.2684 seconds) respectively.

With 12 projections, the barcode generation takes 231.7333221435546875 ms (0.2317 seconds). The time to generate the matches for the entirety of the dataset is 673.38228225708007813 ms (0.6734 seconds), and the time to generate these matches and save them to the test computer is 18359.216213226318359 ms (18.3592 seconds).

The time to generate a single match with 12 projections for dataset index 1, 50, and 100 were 13.066530227661132813 ms (0.0131 seconds), 18.4078216552734375 ms (0.0184 seconds), and 15.038728713989257813 ms (0.0151 seconds) respectively.

The time to generate a single match and save it to the test computer with 12 projections for dataset index 1, 50, and 100 were 244.56787109375 ms (0.2446 seconds), 218.28866004943847656 ms (0.2183 seconds), and 216.907501220703125 ms (0.2169 seconds) respectively.

From the Sample Successful Matches, the similarity between projections can be seen in the figure below:

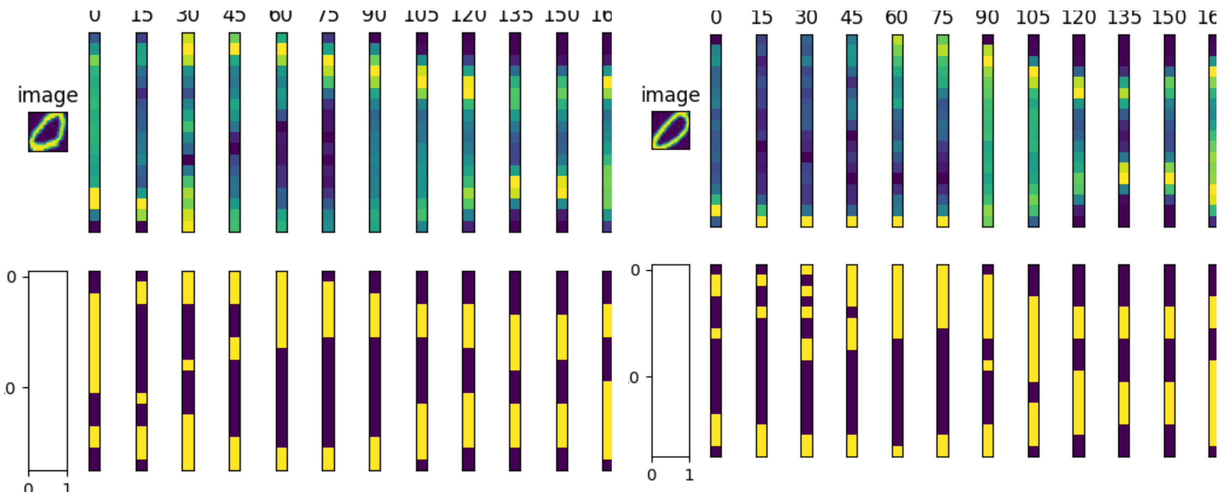


Figure 1: Each projection taken for the first sample successful match provided in Sample Search Results.

The same pattern can be seen in the projections for this Sample Unsuccessful Match:

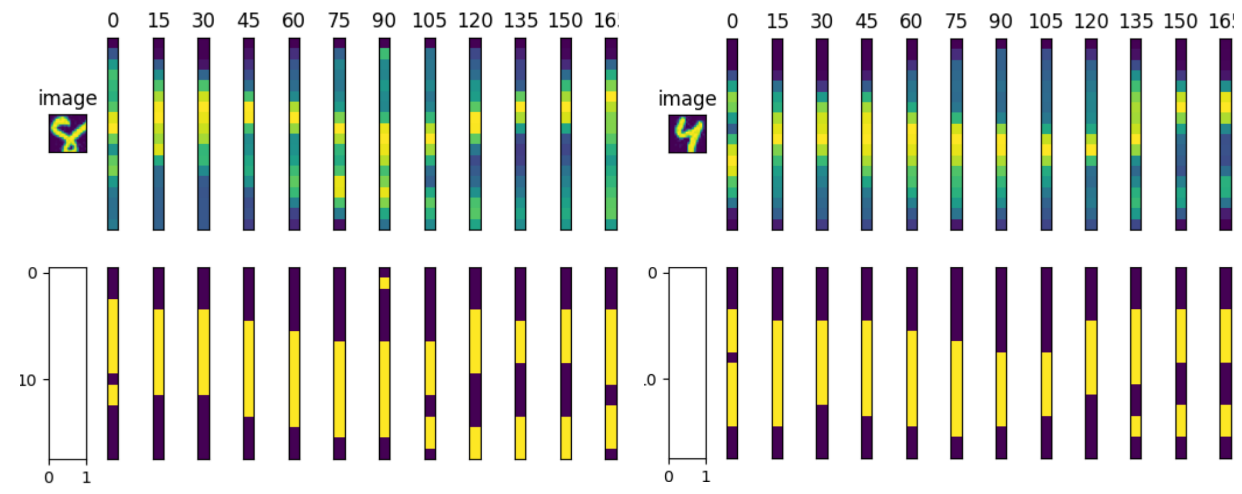


Figure 2: Each projection taken for the last sample unsuccessful match provided in Sample Search Results.

Figures 1 and 2 make it very easy to see how a possible match was made, and offers insight into what the program is "seeing."

## Sample Search Results

### Sample Successful Matches:

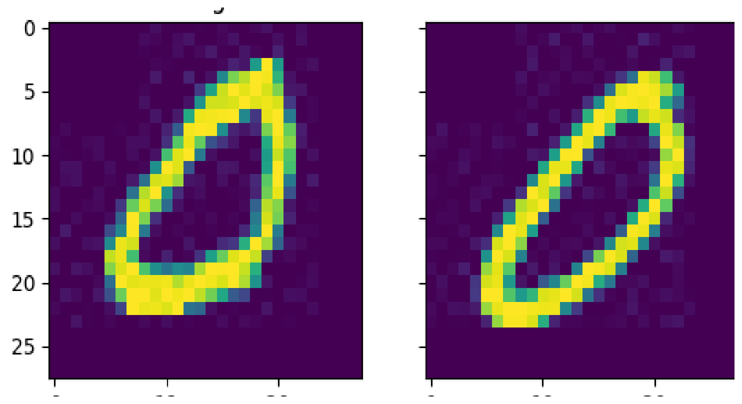


Figure 3: First sample successful match (zero class)

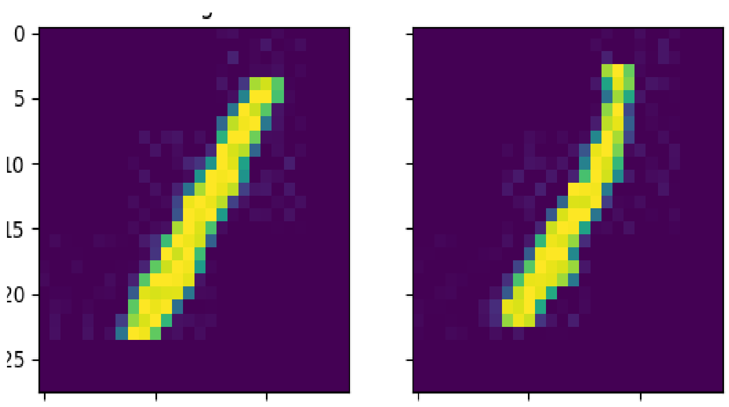


Figure 4: Second sample successful match (one class)

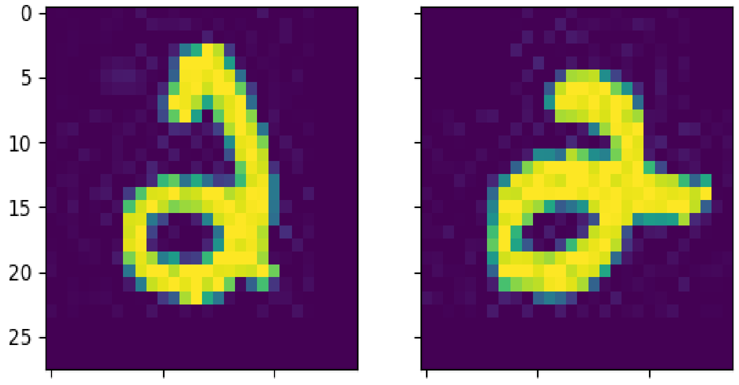


Figure 5: Third sample successful match (two class)

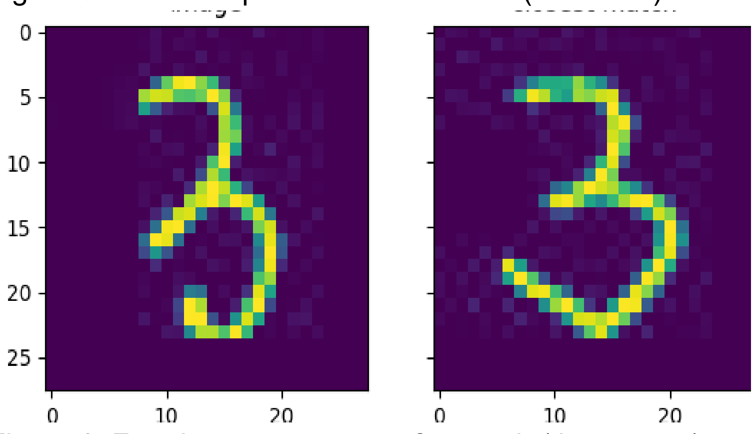


Figure 6: Fourth sample successful match (three class)

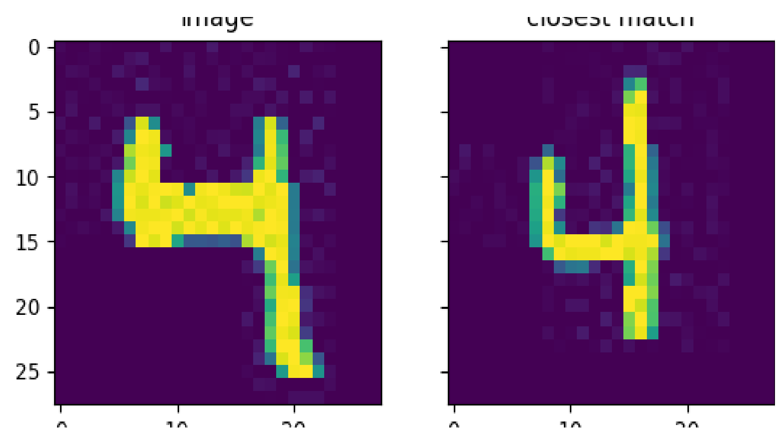


Figure 7: Fifth sample successful match (four class)

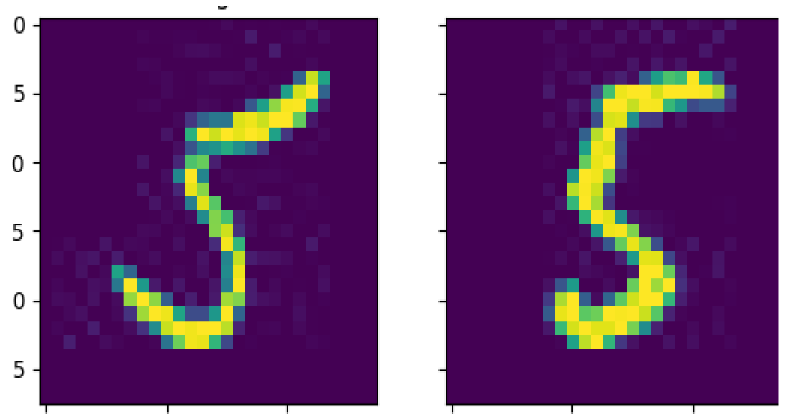


Figure 8: Sixth sample successful match (five class)

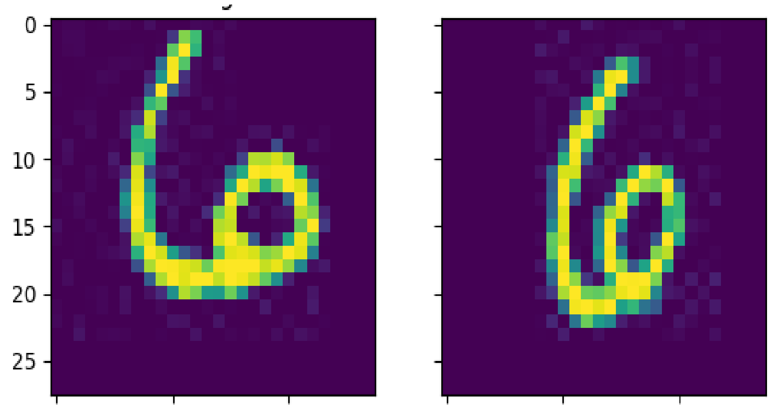


Figure 9: Seventh sample successful match (six class)

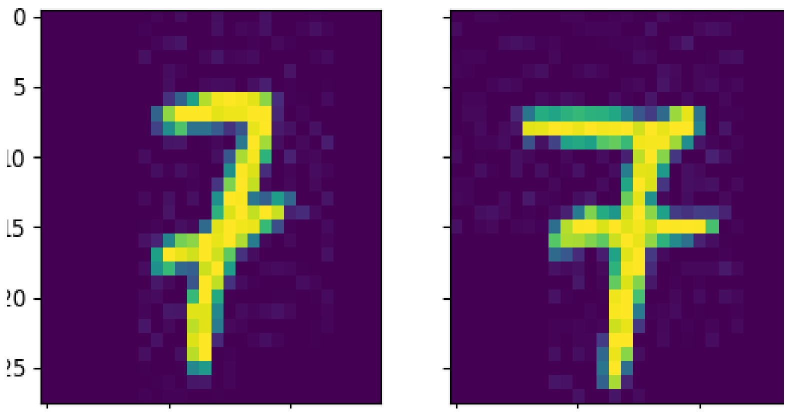


Figure 10: Eighth sample successful match (seven class)

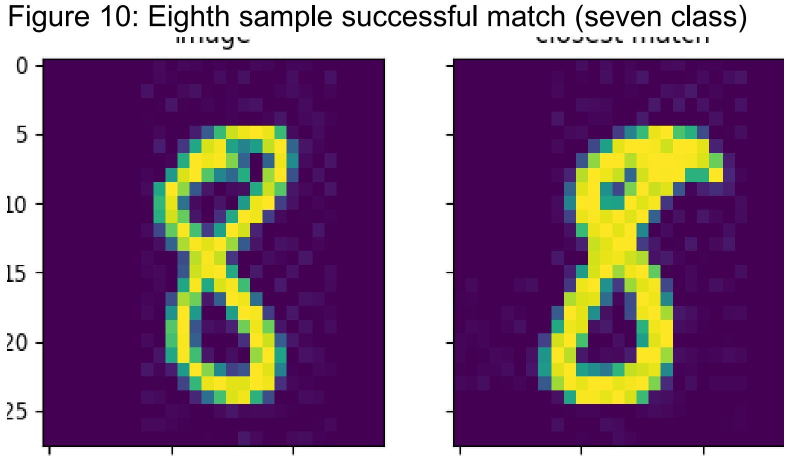


Figure 11: Ninth sample successful  match (eight class)

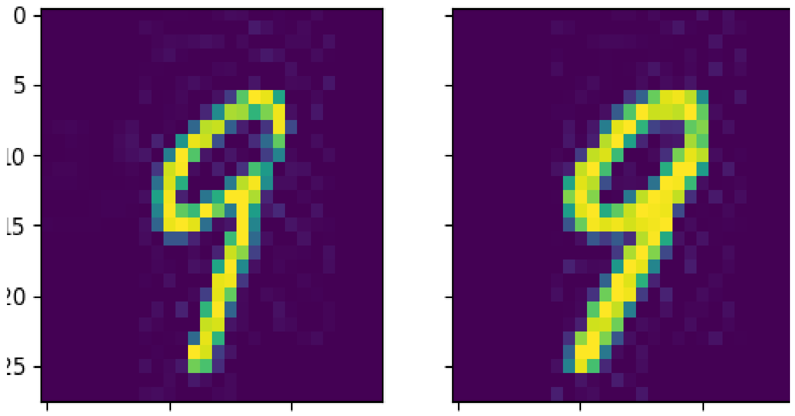


Figure 12: Tenth sample successful match (two class)

## Sample Unsuccessful Matches:

Images matched:

Possible reasoning:

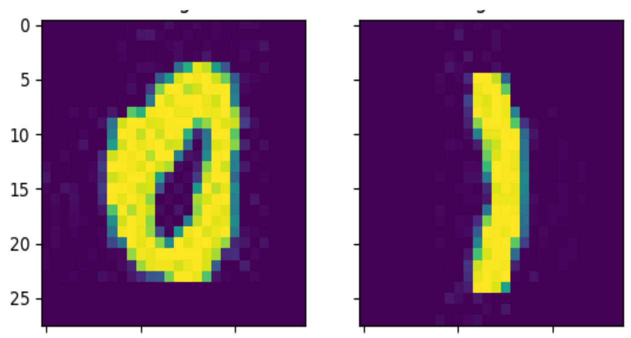


Figure 13: First sample unsuccessful match (zero class)

Handwritten digits had very thick strokes, leading to very high pixel density. Match found was the next “thickest” digit in the data set.

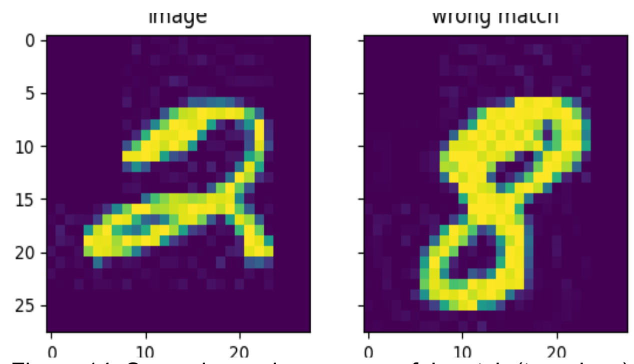


Figure 14: Second sample unsuccessful match (two class)

Bottom loop of two is very similar in structure to the 8. High pixel density in the upper left corner for both numbers. Match is understandable.

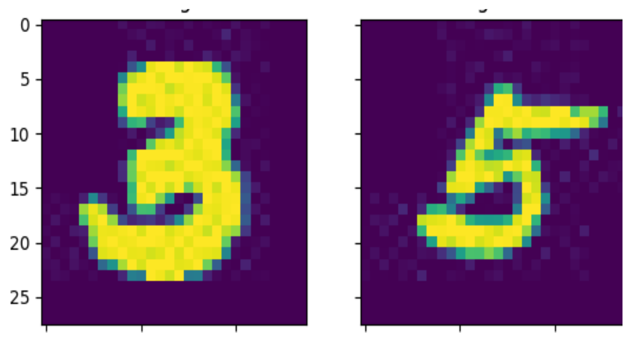


Figure 15: Third sample unsuccessful match (three class)

Query image is very thick, making the overall shape difficult to distinguish and similar to that of the match found. Understandable match.

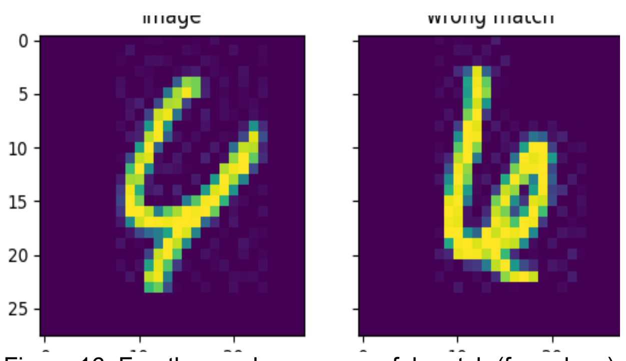


Figure 16: Fourth sample unsuccessful match (four class)

Both the query image and the match found share a similar footprint. The match is understandable.

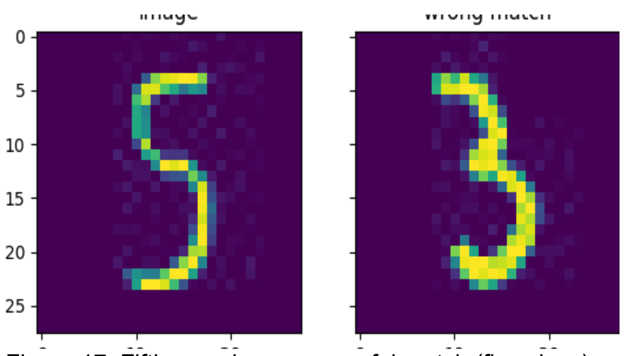


Figure 17: Fifth sample unsuccessful match (five class)

Both of the images are very narrow and share mostly the same footprint. The match makes sense.

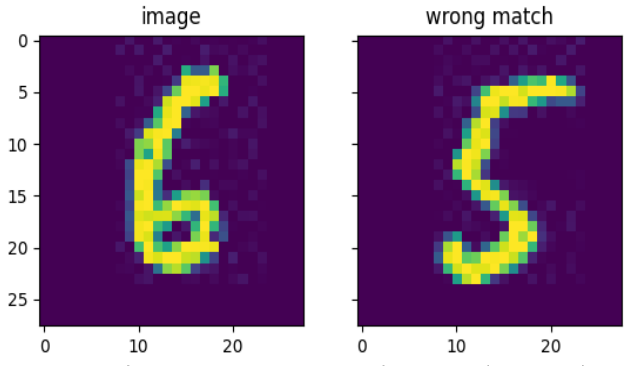


Figure 18: Sixth sample unsuccessful match (six class)

Images share mostly the same footprint, match is understandable.

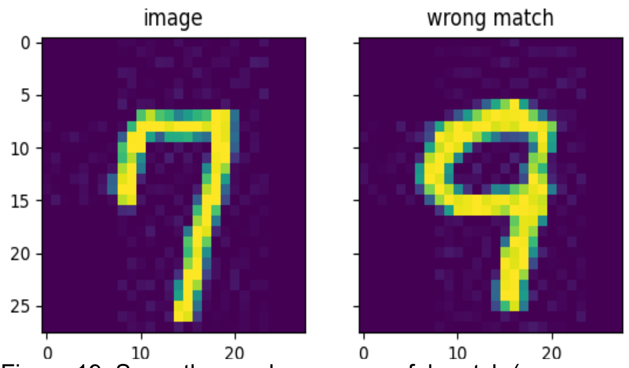


Figure 19: Seventh sample unsuccessful match (seven class)

Images follow a similar structure. Match makes sense. Only one other 7 had a hanging “lip” similar to this one, but it also had a horizontal stroke through the middle so it is understandable this pairing was not made.

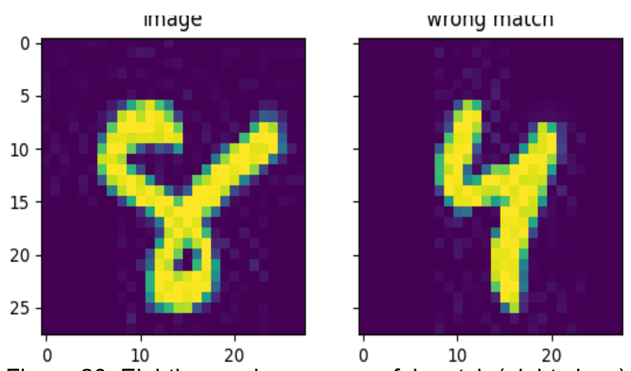
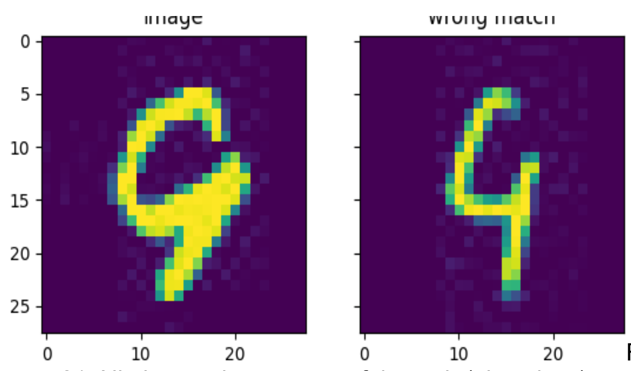


Figure 20: Eighth sample unsuccessful match (eight class)

Both of the numbers are composed of incomplete strokes, and have very high pixel density. They also follow a very similar footprint.



Fi
gure 21: Ninth sample unsuccessful match (nine class)

Both the query image and the found image share VERY similar footprints. It is understandable that this match was made.

## Concluding Remarks

Overall, the program operates successfully. It runs with a hit accuracy of 73% and runs almost equally as well with the addition of newly created images. The program runs in O(n) time to find a match for a single image in the database, and runs in O(n$^2$) to find matches for all of the images in the database.

Searching using the original images, un-cropped, using the 4 recommended projections 0, 45, 90, 135 degrees resulted in a hit ratio of 51%. Uncropped images with 12 projections had a hit ratio of 64% (from Table 2).

Cropping the image to only use rows and columns 4 to 22 to eliminate the empty space that made a border around the images, and increasing the projections to 12 allowed for the hit ratio of the program to increase 22%.

Cropping played the most significant role in improving the hit ratio, as it led to an increase of 17% (51% to 68%) with 4 projections contrasted to the 13% hit ratio increase seen by increasing the projections.

Voting was initially introduced to help further increase the hit accuracy, but this was removed as the hamming distances across the numerous different results varied greatly, and often the closest match was the number returned even if it did not correspond to the digit's label. The varying hamming distances resulted in voting not yielding enough accurate contenders without introducing even more contenders that were incorrect matches, leading to more overall incorrect matches as a whole.

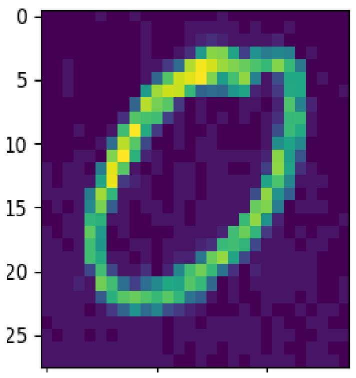
Introducing new hand drawn images into the data set yielded similar results. After the inclusion of 10 new zeroes (drawn on paper, scanned, then colour inverted), the hit accuracy only decreased to 72%, as 8 out of 10 zeroes were matched correctly. See Appendix A for the images introduced and their matches.

One key factor which may have contributed to a reduced hit ratio is the format in which the images were provided. As they are jpeg images, their compression leads to discoloured pixels surrounding the main image, called artifacts. Using an alternative file format, such as gzip (the format of the original dataset), or png could have eliminated the possibility of "false positives" in the radon techniques used to generate the barcode from the projections. Alternatively, some preimage processing could have been done to reduce the noise in the images and filter out these artifacts before attempting to take projections.

To further improve the program, one might consider adding even more projections. While this slows down the overall speed at which it runs due to each barcode increasing by 18 digits for every projection. Currently, each barcode is 216 digits long. When each image was processed using only 4 projections, the accuracy is 68% with barcodes that are 72 digits long.

12 projections were decided upon as it most thoroughly covered the image, and yielded a reasonable increase in accuracy with a relatively small increase in running time as seen in the running time analysis section above and Table 2. While tripling the projections did lead to an increase in running time by a factor of 3, the increase in accuracy is valuable enough to warrant it for a dataset of this size. It is for this reason that 12 projections were utilized. If the dataset were to increase in size significantly the programmer may decide that the trade off is not worth it.

## Appendix A



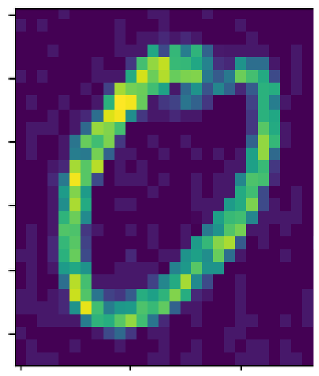


Figure 22: Hand-drawn zero on left matched with zero drawn by the same person on right.

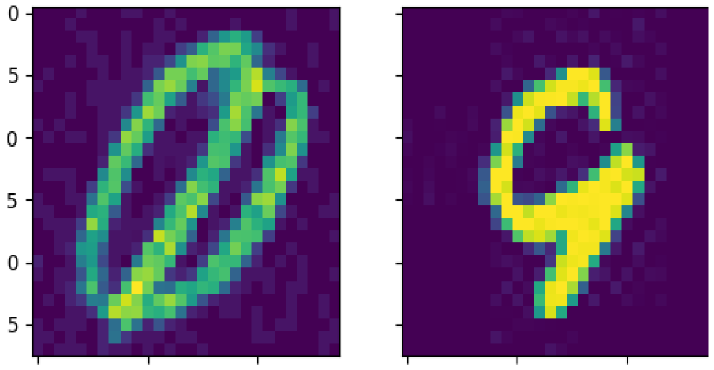


Figure 23: Hand-drawn zero on left matched unsuccessfully with 9 from dataset.

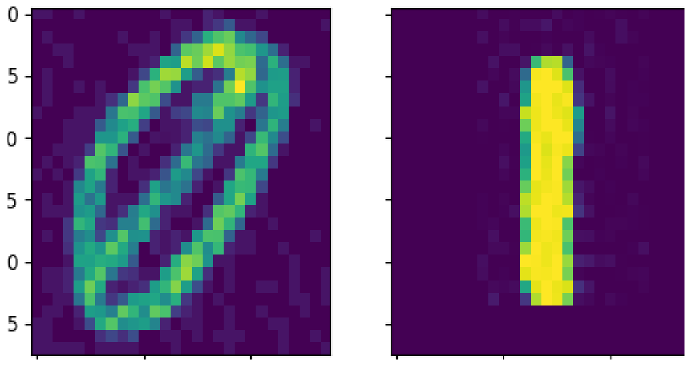


Figure 24: Hand-drawn zero on left matched unsuccessfully with 1 from dataset.

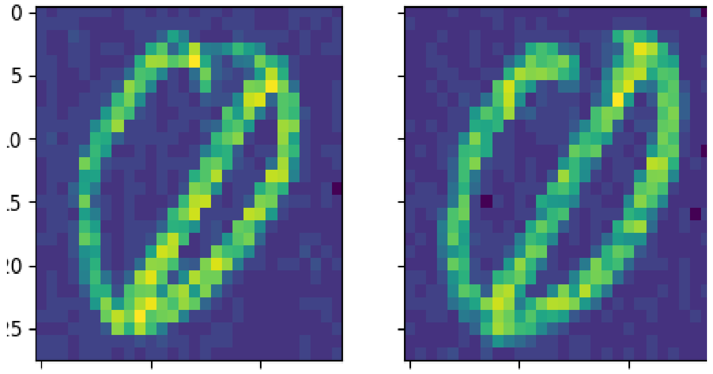


Figure 25: Hand-drawn zero on left matched with zero drawn by the same person on right.

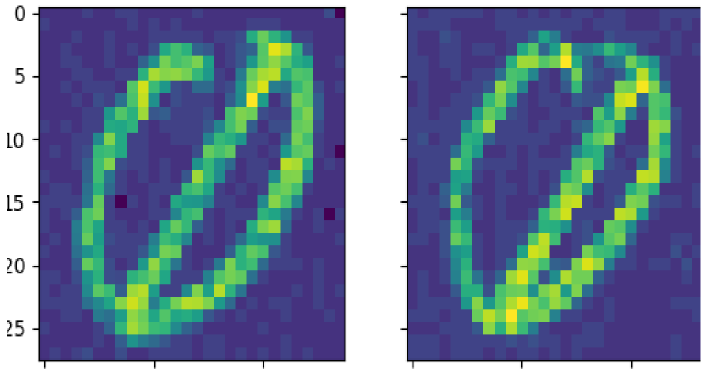


Figure 26: Hand-drawn zero on left matched with zero drawn by the same person on right.

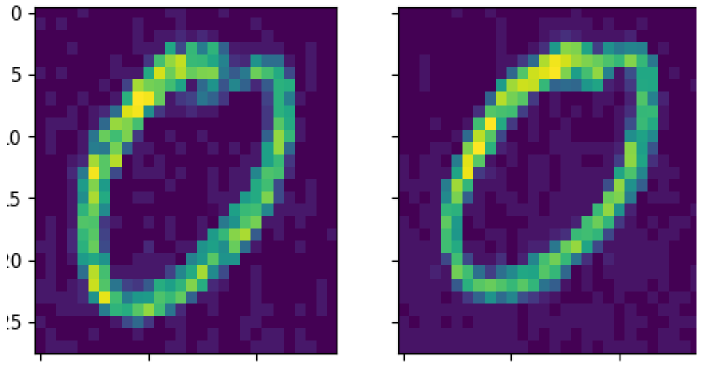


Figure 27: Hand-drawn zero on left matched with zero drawn by the same person on right.

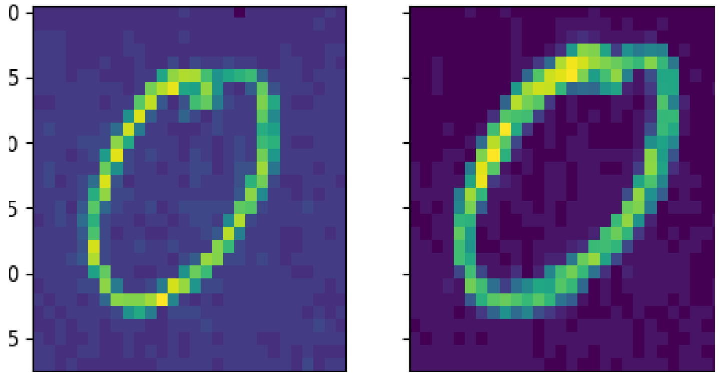


Figure 28: Hand-drawn zero on left matched with zero drawn by the same person on right.

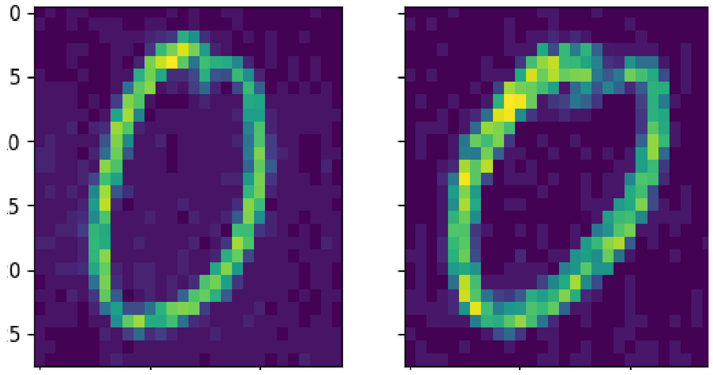


Figure 29: Hand-drawn zero on left matched with zero drawn by the same person on right.

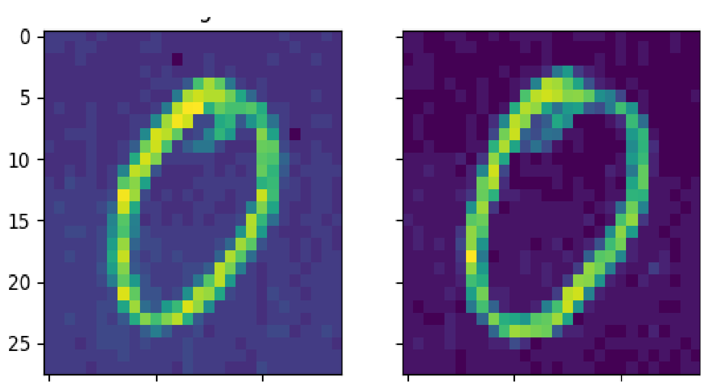


Figure 30: Hand-drawn zero on left matched with zero drawn by the same person on right.

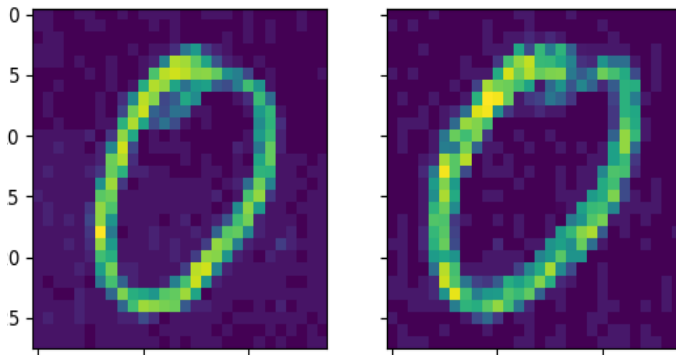


Figure 31: Hand-drawn zero on left matched with zero drawn by the same person on right.