

Content Based Image Retrieval Using **Barcodes**

By

**Muhmamad Zahid (100711463), Thomas Menegotti(100750648),
Matheeshan Sivalingam(100703887), and Harveen
Sandhu(100704514)**

Introduction

Content based image retrieval (CBIR) is a method of finding similar images by comparing the content of an input image to the content of the images in the dataset. In this methodology the software receives an input image, extracts information from the content of the input image and uses that extracted information to find the most similar images. The developers of this project (group 16) decided to develop it in Python. To successfully implement our vision we used quite a few libraries including pillow, numpy, and PyQt5.

Algorithm Explanation and Complexity Analysis

There are two main algorithms used in the design of this project. The first algorithm is the “barcode generator” algorithm.

```
def barcodeGenerator(self):
    barcode = ""
    n = self.imgArray.shape[0]
    self.P["P1"] = np.zeros(n)
    self.P["P2"] = np.zeros(2*n-1)
    self.P["P3"] = np.zeros(n)
    self.P["P4"] = np.zeros(2*n-1)
    val = 0
    for i in range(n):
        for j in range(n):
            self.P["P1"][i] = self.P["P1"][i] + self.imgArray[i][j]

    for i in range(n):
        for j in range(n):
            self.P["P2"][i-j+n-1] = self.P["P2"][i-j+n-1] + self.imgArray[i][j]

    for i in range(n):
        for j in range(n):
            self.P["P3"][j] = self.P["P3"][j] + self.imgArray[i][j]

    for i in range(n):
        for j in range(n):
            self.P["P4"][i+j] = self.P["P4"][i+j] + self.imgArray[i][j]

    self.thresholds["P1"] = sum(self.P["P1"])/len(self.P["P1"])
    self.thresholds["P2"] = sum(self.P["P2"])/len(self.P["P2"])
    self.thresholds["P3"] = sum(self.P["P3"])/len(self.P["P3"])
    self.thresholds["P4"] = sum(self.P["P4"])/len(self.P["P4"])

    for projaxis in ["P1", "P2", "P3", "P4"]:
        values = self.P[projaxis]
        for proj in values:
            if proj > self.thresholds[projaxis]:
                barcode = barcode + "1"
            else:
                barcode = barcode + "0"
        self.bitString += barcode
    barcode = ""
```

Fig. 1: BarcodeGenerator Algorithm

Given an input image, the BarcodeGenerator algorithm creates 4 arrays representing the projections. The projections are along the horizontal, diagonal, vertical and other diagonal of the image. After extracting

the projections the algorithm finds the threshold value (which is the average value of the projection) and uses that to binarize the projection. For each projection, and each value, if the value is greater than the threshold then it is appended as “1” to the barcode. Otherwise “0” is appended to the barcode.

Time Complexity of BarcodeGenerator:

The time complexity of this algorithm is $O(n^2)$ due to the nested for loops.

The second algorithm used is the searching algorithm.

```
def search(dataset, image, threshold=23):
    listAll = defaultdict(list)
    correct, wrong = 0, 0
    # print("Search")
    for i in dataset:
        x = MNISTImage.hammingDistance(i, image)
        # print(i.bitString)
        if x < threshold:
            if image.actualCharacter == i.actualCharacter:
                correct += 1
            else:
                wrong += 1
    if correct + wrong == 0:
        return 0
    else:
        accuracy = correct * 100 / (correct + wrong)
    return accuracy

def searchImages(dataset, image, threshold=23):
    imagePaths = []
    for i in dataset:
        x = MNISTImage.hammingDistance(i, image)
        # print(i.imgPath, x)
        if x < threshold and image.imgPath != i.imgPath:
            # print("MATCH")
            # print(i.imgPath)
            imagePaths.append(i.imgPath)
    return imagePaths
```

Fig. 2: Searching Algorithm

Given an input image, this algorithm loops over the entire dataset and finds the hamming distance between the input and the images in the dataset. If the hamming distance is less than the threshold specified, then the program counts that as a hit, and for all images where the hamming distance is greater than the threshold, then that is counted as a miss. The program returns the accuracy as: the numbers of hits divided by the total number of images in the dataset.

```
./MNIST_DS/0/img_10007.jpg : 12.86 %  
./MNIST_DS/0/img_10010.jpg : 18.37 %  
./MNIST_DS/0/img_10017.jpg : 19.15 %  
./MNIST_DS/0/img_10032.jpg : 12.33 %  
./MNIST_DS/0/img_10039.jpg : 26.47 %  
./MNIST_DS/0/img_10123.jpg : 26.67 %  
./MNIST_DS/0/img_10156.jpg : 31.82 %  
./MNIST_DS/0/img_10207.jpg : 12.0 %  
./MNIST_DS/0/img_10231.jpg : 25.81 %  
./MNIST_DS/0/img_10242.jpg : 13.43 %  
./MNIST_DS/1/img_1000.jpg : 16.67 %  
./MNIST_DS/1/img_10006.jpg : 14.71 %  
./MNIST_DS/1/img_10009.jpg : 14.04 %  
./MNIST_DS/1/img_10040.jpg : 13.95 %  
./MNIST_DS/1/img_10042.jpg : 14.29 %  
./MNIST_DS/1/img_10076.jpg : 20.0 %  
./MNIST_DS/1/img_10175.jpg : 14.29 %  
./MNIST_DS/1/img_10189.jpg : 18.18 %  
./MNIST_DS/1/img_10190.jpg : 18.75 %  
./MNIST_DS/1/img_10192.jpg : 17.39 %  
./MNIST_DS/2/img_10000.jpg : 7.58 %  
./MNIST_DS/2/img_10027.jpg : 8.06 %  
./MNIST_DS/2/img_10031.jpg : 12.0 %  
./MNIST_DS/2/img_10062.jpg : 16.67 %  
./MNIST_DS/2/img_10068.jpg : 33.33 %  
./MNIST_DS/2/img_1011.jpg : 8.82 %  
./MNIST_DS/2/img_10111.jpg : 7.04 %  
./MNIST_DS/2/img_1020.jpg : 0.0 %  
./MNIST_DS/2/img_10285.jpg : 8.33 %  
./MNIST_DS/2/img_10297.jpg : 30.77 %  
./MNIST_DS/3/img_10016.jpg : 33.33 %  
./MNIST_DS/3/img_10029.jpg : 17.14 %  
./MNIST_DS/3/img_10045.jpg : 29.41 %  
./MNIST_DS/3/img_10049.jpg : 60.0 %  
./MNIST_DS/3/img_10117.jpg : 13.16 %  
./MNIST_DS/3/img_10121.jpg : 25.0 %  
./MNIST_DS/3/img_10140.jpg : 10.0 %  
./MNIST_DS/3/img_10179.jpg : 7.81 %  
./MNIST_DS/3/img_10193.jpg : 27.78 %
```

Fig. 3: Accuracy Results

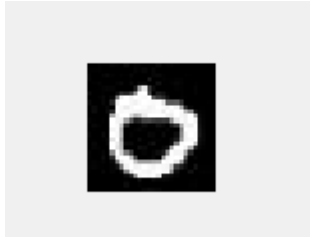
Time Complexity of BarcodeGenerator:

The time complexity of this algorithm is $O(nm)$ where n is defined as the number of images in the dataset and m is the length of the generated barcodes of the images.

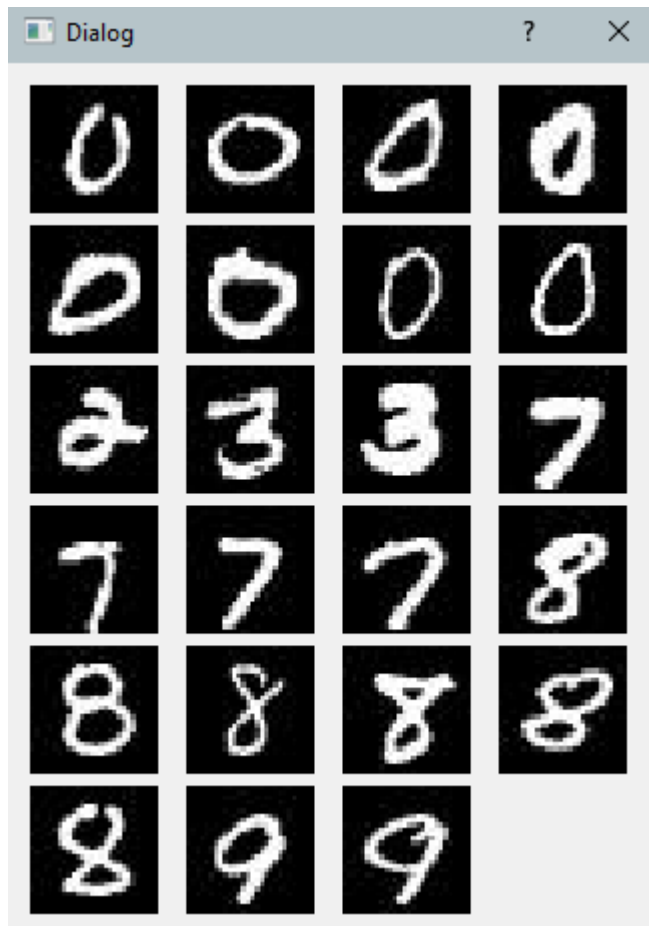
Search Results

Class 0

Input:



Results:

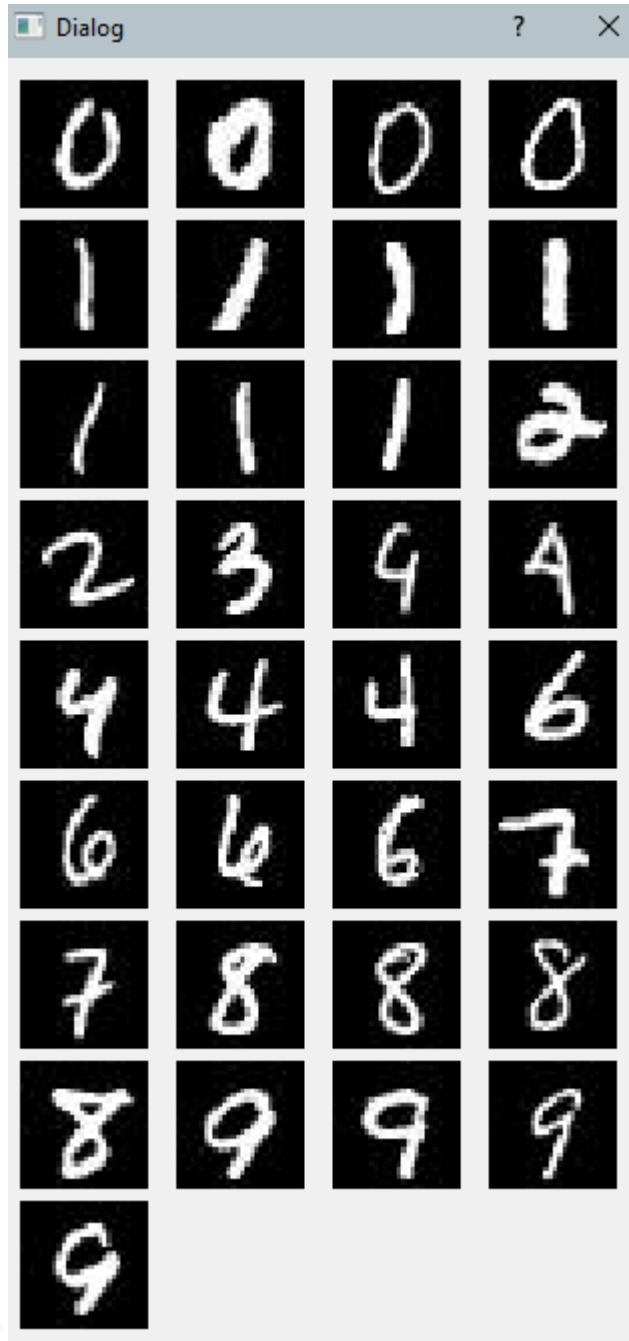


Class 1

Input:



Results:

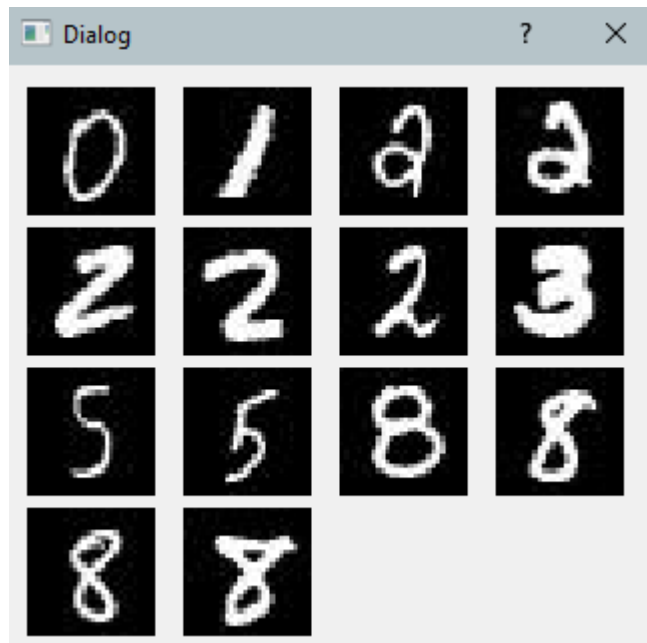


Class 2

Input:



Results:



Class 3

Input:



Results:

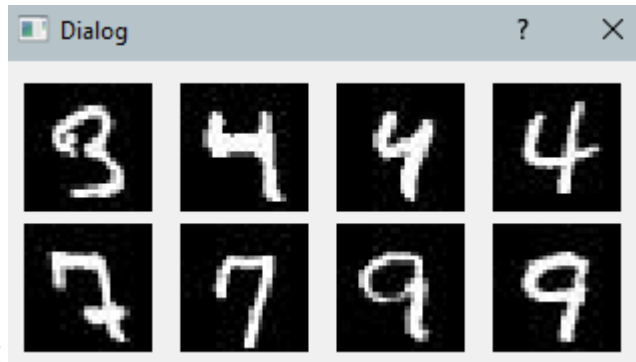


Class 4

Input:



Results:

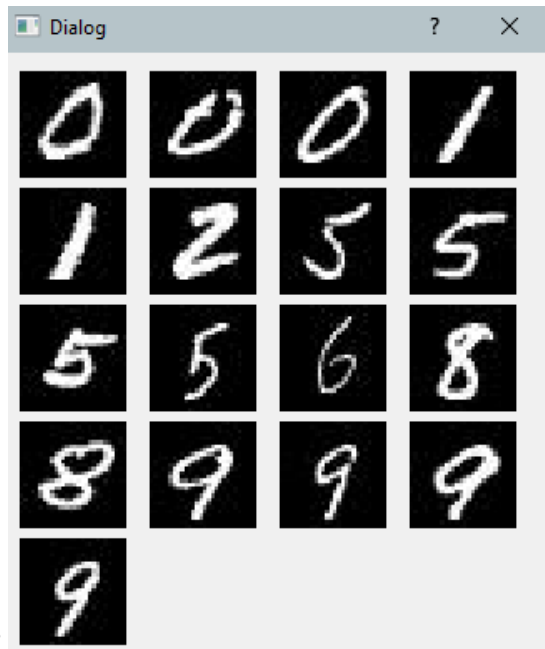


Class 5

Input:




Results:



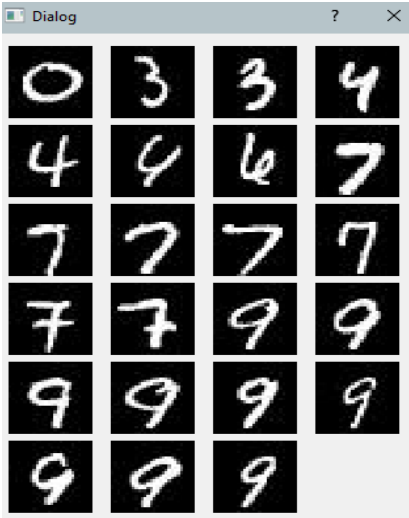
Class 6

Input: 

Results: 

Class 7

Input: 

Results: 

Class 8

Input:



Results:



Class 9

Input:



Results:



Conclusion

CBIR is an important field that will have huge impacts in many areas (such as medical imaging, astronomy, law enforcement) in the future. Our CBIR program just barely scratches the surface of the potential in this methodology. While our program was able to retrieve similar images from the dataset given an input image, there are still many areas where it can improve. The first area of improvement is accuracy. We can potentially increase accuracy by creating more projections. We estimate that by doubling the projections, the accuracy for each image should increase by that factor as well. Another area for improvement is in the time complexity for the algorithms. Since our dataset is relatively small the amount of time the algorithms take to process is not too much. However with a dataset of let's say one million, the amount of time the algorithms would take to process would be staggering. We should aim for a time complexity of at most $O(\log n)$. We believe we would be able to achieve this time complexity had we implemented some parts of the algorithms recursively rather than iteratively.