Pattern Recognition Final Assignment: Handwritten Digits Classification using Shape Features

Arnes Respati Putri
16/392817/PA/17084

Faculty of Mathematics and Natural Sciences
Universitas Gadjah Mada

May 2019

1 Introduction

In image processing, feature extraction plays a significant role. Prior to getting the features, the images have went through several preprocessing steps that are taken accordingly to the goal of the image processing. When extracting the features, it is necessary to apply the appropriate techniques.

Features are the unique signatures or properties of a certain image that defines an image and therefore can differentiate one image from another. An image consists of pixels. Considering each pixel can have an 8-bit value, a moderate-size image of dimension 640 x 480 will have 2,457,600 bits of information. That said, in feature extraction, it is necessary to derive features that are informative and non-redundant, providing dimensionality reduction and thereby computational complexity reduction. The lowering of the dimension is also helps leading to better human interpretations, resulting in high understanding of the model that can be advantageous to the time taken to do the image processing. Hence, feature extraction can simplify and ease the subsequent processing steps and the result of the classification will be optimal.

In the classification of handwritten digit images in this report, the shape features are used. The implementation is done in Python, where the libraries **numpy**, **opencv cv2**, and **scikit-learn** are used.

2 Data Acquisition

There are 30 images used for the training data and 5 images used for testing data in each class, 0 to 9. The data are taken from the ARDIS (Arkiv Digital Sweden) dataset[1]. The images in ARDIS dataset are extracted from 15.000

Swedish church records which were written by different priests with various handwriting styles in the nineteenth and twentieth centuries. The used dataset is the ARDIS dataset III which contains 7600 handwritten digit images with clean background. The figure below illustrates handwritten digit images from dataset III in ARDIS.

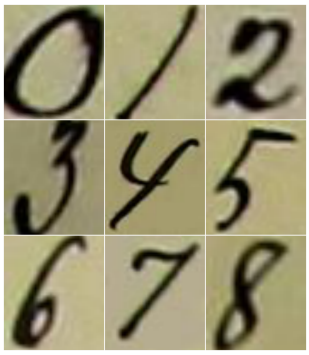

Figure 1. ARDIS original image.

The original images dataset are of different dimensions, ranging from 20×20 to 40×40. These images are resized to be the same size, which is 30×30 to make the data uniform.

3 Image Preprocessing

3.1 Binarization

Initially, the image dataset acquired from the data repository are those of RGB (red, green, blue) color. Image binarization takes the pixel intensity representing either red, green, or blue color of the image and replace it with the values of either 0 and 1. Zero represents the color black and one represents the color white. The binarization reduce the computation cost as it only considers two colors of the image. Below is the source code for the binarization.

```
def binarize_image (image, threshold_list):
 minv = np.amin(image)
 maxv = np.amax(image)
 height, width = image.shape[:2]

 # Using otsu method to find threshold
 # Histogram of color value
 ret,thresh = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.
    THRESH_OTSU)

 threshold_list.append(thresh)

 for i in range(0, height):
  for j in range(0 , width):
       if image[i][j] > ret:
        image[i][j] = 0
```

```
16          else:
17           image[i][j] = 1
```

Listing 1: Binarization of image.

The main idea of the binarization is if the value of the pixel is greater than a certain threshold, it will be converted to black (represented by 0), and otherwise white (represented by 1). The threshold for each image is obtained based on the distribution of the pixel values, i.e the color histogram. The method that is used in this implementation is the Otsu method. After computing the binarization, the result is get as follows.

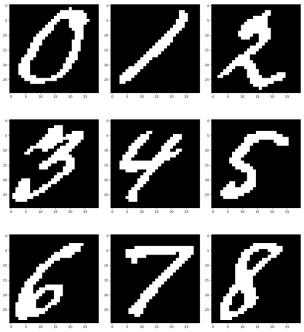

Figure 2. Binarized image.

4 Feature Extraction

The features that are being extracted from the images are the based on the shape and the skeleton of the image. The shape measurements include area, perimeter, and compactness, whereas the skeleton measurement include the number of holes.

4.1 Boundary Based Measurement

The area of an object can be determined by counting all the pixels contained in the object. The perimeter is the total length of the object boundary. The perimeter can be measured by tracing the boundary of the object and summing all the steps. Compactness is calculated in respect to the object being a circle, which it will have the value of 1. Mathematically, the compactness is written as:

$$c = \frac{l^2}{4\pi A}$$

where l is the perimeter, and A the object area. The factor 4π ensures that c equals 1 (the lowest possible value) for a circle. The listings below depicts the source code of the calculations.

```
def calc_compactness(area, perimeter, mlist):
 compactness = perimeter**2 / 4 * 3.14 * area
 mlist.append(compactness)
```

Listing 2: Calculating the compactness of the object.

```
def calc_area(image, mlist):
 height, width = image.shape[:2]
 area = 0
 for i in range(0, height):
  for j in range(0 , width):
   if image[i][j] == 1 :
    area += 1
       # area = counting all the pixels contained in the object
 mlist.append(area)
```

Listing 3: Calculating the area of the object.

When calculating the perimeter of the object, it is necessary to find the edge of the object. This can be done by applying Sobel operation. The Sobel Operator uses two kernels (one for each direction):

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

We compute the convolution between the image (converted in black and white) and the two kernels separately. That gives each pixel the values mag_x and mag_y. The value of the current pixel is set at:

$$\sqrt{(mag_x)^2 + (mag_y)^2}$$

The following is the source code of the sobel operation and the calculation of the perimeter.

```
def find_edge(image, mlist):
 height, width = image.shape[:2]

 new_image = [[[] for col in range (0,width)] for row in range(0,
    height)]
 kernel_x = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
 kernel_y = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]]

 segmented_pixels = [[0,0,0],[0,0,0],[0,0,0]]

 perimeter = 0

 for i in range(0, height):
  for j in range(0 , width):
   # Find index neighbour
   segmented_pixels = neighbors(image, i, j, 1)

```

```
    res_x = matrix_multiplication(segmented_pixels, kernel_x)
    res_y = matrix_multiplication(segmented_pixels, kernel_y)

    val = int(sqrt((res_x[1][1]**2) + (res_y[1][1]**2)))

    new_image[i][j] = int(val/2)

    if new_image[i][j] >= 1:
     perimeter +=1

image = new_image
# show_image(image)
mlist.append(perimeter)
```

Listing 4: Calculating the perimeter of the object.

After running the above code of the sobel operation, we get the result as follows.

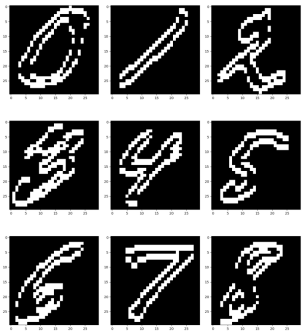

Figure 3. Image object edges.

4.2 Skeleton Based Measurement

The number of holes inside an object is an important topological measure, especially for recognizing digits. For example, the digit 0 has 1 hole, the digit 1 has 0 hole, and the digit 8 has 2 holes. Given a skeleton of an object, the number of holes can be determined by:

$$h = 1 + \frac{b - e}{2}$$

where h is the number of holes, and b and e are the number of branches and endpoints of the skeleton. The skeleton of the object is obtained by using the library **skeletonize** from **skimage.morphology**. Skeletonizing basically calculating the medial axis, or the set of all points having more than one closest point on the object's boundary. The skeleton of the digit images are depicted in the following image.

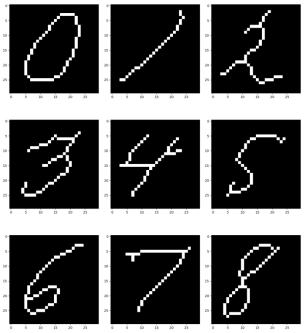

Figure 4. Image object skeleton.

```
def count_no_of_holes(image, mlist):
 no_branches = count_intersection(image)
 no_endpoints = count_end_points(image)

 holes = 1 + ((no_branches - no_endpoints)/2)
 holes = int(holes)

 mlist.append(holes)
```

Listing 5: Counting the number of holes.

```
def count_intersection(image):
 height, width = image.shape[:2]
 skeleton_image = skeletonize(image).astype(np.uint8)
    return_distance=True)
 intersections = list()
 neighbors_matrix = [[0,0,0],[0,0,0],[0,0,0]]

    # List of possible intersection combination
 validIntersection = [[0,1,0,1,0,0,1,0],[0,0,1,0,1,0,0,1],
    [1,0,0,1,0,1,0,0], [0,1,0,0,1,0,1,0],
    [0,0,1,0,0,1,0,1],[1,0,0,1,0,0,1,0],
    [0,1,0,0,1,0,0,1],[1,0,1,0,0,1,0,0],
    [0,1,0,0,0,1,0,1],[0,1,0,1,0,0,0,1],
    [0,1,0,1,0,1,0,0],[0,0,0,1,0,1,0,1],
    [1,0,1,0,0,0,1,0],[1,0,1,0,1,0,0,0],
    [0,0,1,0,1,0,1,0],[1,0,0,0,1,0,1,0],
    [1,0,0,1,1,1,0,0],[0,0,1,0,0,1,1,1],
    [1,1,0,0,1,0,0,1],[0,1,1,1,0,0,1,0],
    [1,0,1,1,0,0,1,0],[1,0,1,0,0,1,1,0],
    [1,0,1,1,0,1,1,0],[0,1,1,0,1,0,1,1],
    [1,1,0,1,1,0,1,0],[1,1,0,0,1,0,1,0],
    [0,1,1,0,1,0,1,0],[0,0,1,0,1,0,1,1],
    [1,0,0,1,1,0,1,0],[1,0,1,0,1,1,0,1],
    [1,0,1,0,1,1,0,0],[1,0,1,0,1,0,0,1],
    [0,1,0,0,1,0,1,1],[0,1,1,0,1,0,0,1],
    [1,1,0,1,0,0,1,0],[0,1,0,1,1,0,1,0],
    [0,0,1,0,1,1,0,1],[1,0,1,0,0,1,0,1],
    [1,0,0,1,0,1,1,0],[1,0,1,1,0,1,0,0]]

 for i in range(0, height):
```

```
  for j in range(0 , width):
   # If white pixel is found
   if skeleton_image[i][j] == 1:
    neighbors_matrix =
                neighbors(skeleton_image, i, j, 1)
    neighbors_matrix =
                np.array(neighbors_matrix)
    neighbors_matrix =
                neighbors_matrix.flatten()
    neighbors_matrix =
                np.delete(neighbors_matrix, 4).tolist()

    valid = True
    if neighbors_matrix in validIntersection:
     intersections.append((j,i))

# Filter intersections to make sure we don't count them twice or
    ones that are very close together
 for point1 in intersections:
  for point2 in intersections:
   if (((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2) <
    10**2) and (point1 != point2):
    intersections.remove(point2)
# Remove duplicates
 intersections = list(set(intersections))
 return len(intersections);
```

Listing 6: Counting the number of intersections.

```
def count_end_points(image):
 height, width = image.shape[:2]
 skeleton_image = skeletonize(image).astype(np.uint8)
 neigh = [[0,0,0],[0,0,0],[0,0,0]]
 counter = 0
 for i in range (0, height):
  for j in range (0, width):
   if skeleton_image[i][j] == 1:
    neigh = neighbors(skeleton_image, i, j , 1)
    if np.sum(neigh) == 2:
     print(i,j)
     counter +=1
 return (counter)
```

Listing 7: Counting the number endpoints.

5 Classification

Based on the feature extraction done as described in the previous section, the number of holes and the compactness of the object in the image will be used for the classification. The input feature is then contrived of two dimensions. Figure 5 and 6 below depict the points of the pictures in 2-dimensional plane.

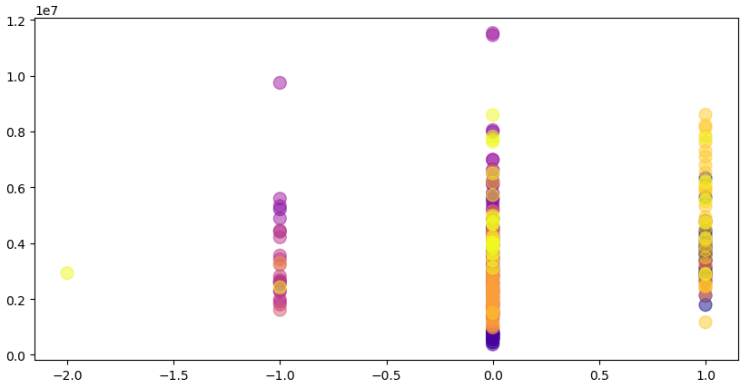

Figure 5. Plotting of training image feature points.

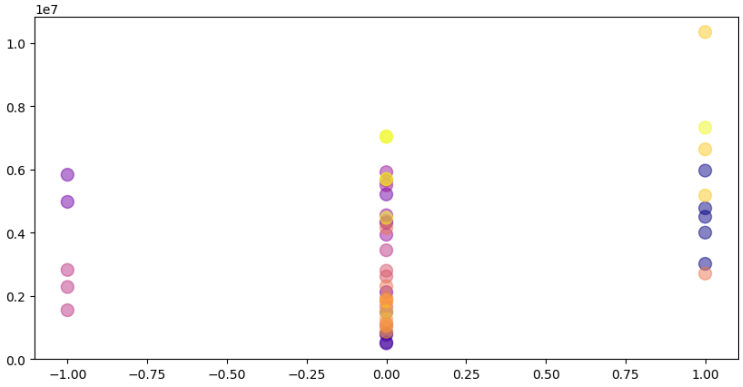

Figure 6. Plotting of testing image feature points.

5.1 Cosine Similarity Matching

The cosine distance calculation is aided by the library **scikit-learn**.

```
from sklearn.metrics.pairwise import cosine_similarity

def find_cosine_similarity(features_1, features_2, label, index):
 row = len(features_2)
 similarity_per_row = [0.0 for i in range (0, row)]

 for r in range(0, row):
  similarity_per_row[r] = 1 - spatial.distance.cosine(features_1,
    features_2[r])

 max_index = similarity_per_row.index(max(similarity_per_row))
 label[index] = max_index
```

Listing 8: Calculating the similarity using cosine distance.

```
Similarity result:  [[0, 0, 0, 0, 8], [1, 1, 1, 5, 1], [1, 5, 4, 4, 1], [1,
2, 1, 1, 1], [4, 4, 1, 2, 4], [1, 1, 1, 1, 1], [1, 1, 1, 0, 1], [1, 1, 1, 1,
 1], [6, 1, 8, 1, 8], [1, 1, 8, 1, 1]]
```

Figure 7. Result of Cosine Similarity Matching.

The accuracy is obtained by the number of correct label prediction, divided by the number of all images.

$$Accuracy = \frac{13}{50} \times 100\%$$

$$Accuracy = 26\%$$

As what depicted in the Figure 7, the label prediction of the image class is 26%. When tried to use 3 features, i.e number of holes, perimeter, and compactness, the accuracy rises to 30% and when added a fourth feature, i.e. area, the accuracy increased to 32%.

5.2 K-Nearest Neighbor

The classification of the image data using K-Nearest Neighbor (KNN) of $K = 5$. The KNN calculation is aided by the library **scikit-learn**.

```
from sklearn.neighbors import KNeighborsClassifier


no_neighbors = 5
knn = KNeighborsClassifier(n_neighbors = no_neighbors)

#Train the model using the training sets
knn.fit(feature_set_train, labels_train)

#Predict the response for test dataset
test_pred = knn.predict(feature_set_test)

# ACCURACY
# Model Accuracy, how often is the classifier correct?
print('Number of neighbors: ', no_neighbors)
print("KNN Accuracy:",metrics.accuracy_score(labels_test, test_pred
    ))
```

Listing 9: Classification using 5 Nearest Neighbor.

```
Number of neighbors:  5
KNN Accuracy: 0.34
```

Figure 8. Result of 5-NN Classification.

$$Accuracy = 0.34 \times 100\%$$

$$Accuracy = 34\%$$

The result of 5-NN classification gives 34% of accuracy. As opposed to the cosine similarity measurement, after adding a third and fourth feature, i.e. the perimeter and the area, the accuracy is still 34%.

6 Conclusion

As opposed to classification using the features of pixel intensity, i.e. color, in classifying 10 handwritten digits, it is more preferred to extract the distinctive features of each digit shape. By extracting the shape features of the digit object, there are new features of the original image that are discovered such as the compactness and the number of holes.

The results of the classification using cosine similarity and KNN for two features, the number of holes and object compactness are 26% and 34%, respectively. The low accuracy result is caused by the different patterns in the handwriting styles in the images. That said, it effects the calculation of the image holes and therefore can make the feature calculation incorrect. Furthermore, after trying to add the other two features, i.e. the perimeter and area, the cosine similarity matching shows an increase of 2% after adding each feature, while the KNN classification accuracy remains the same. It can be inferred that adding the area and the perimeter as additional features is treated redundant in the KNN classification.

References

[1] Huseyin Kusetogullari et al. "ARDIS: a Swedish historical handwritten digit dataset". In: *Neural Computing and Applications* (2019). ISSN: 09410643. DOI: 10.1007/s00521-019-04163-3.