



ME5405 Machine Vision

Group1 Project Report

AY2023/2024

Student Name: Zhang Zhen A0285019B
 Hu Junjie A0285146Y
 Huang Yueteng A0285268N
Supervisor: Chui Chee Kong
 Guillaume Adrien Sartoretti

Content

1. Problem Statement and Solution	2
1.1 Problem Statement.....	2
1.2. A Brief Description of the Solution.....	4
2. Implementation process.....	7
2.1 Binarize.....	7
2.2 rotate	11
2.3 Edge.....	13
2.4 Thin	15
2.5 Label.....	18
2.6 RgbToGray	20
3. Machine Learning Methods.....	21
3.1 Supervised Classification	21
4. Appendix	38

3. Determine an one-pixel thin image of the objects.
4. Determine the outline(s).
5. Label the different objects.
6. Rotate the original image by 30 degrees, 60 degrees and 90 degrees respectively.

1.1.2 Image 2: hello_world.jpg

Image 2

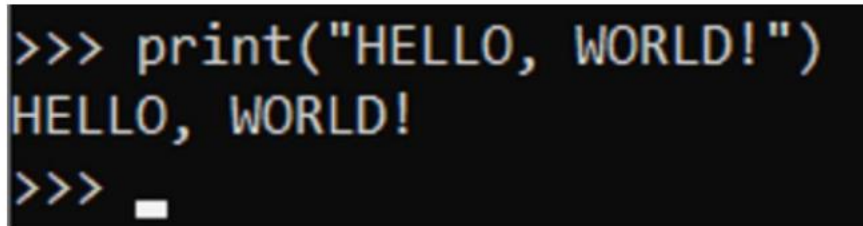


Figure 1-2 hello_world.jpg

Image 2 (Figure 1-2) is a JPEG color image featuring three lines of characters. The objective is to perform multiple tasks on this image.

1. Display the original image on screen.
2. Create an image which is a sub-image of the original image comprising the middle line – HELLO, WORLD.
3. Create a binary image from Step 2 using thresholding.
4. Determine a one-pixel thin image of the characters.
5. Determine the outline(s) of characters of the image.
6. Segment the image to separate and label the different characters.
7. Using the training dataset provided on LumiNUS (p_dataset_26.zip), train the (conventional) unsupervised classification method of your choice (i.e., self-ordered maps (SOM), k-nearest neighbors (kNN), or support vector machine (SVM)) to recognize the different characters (“H”, “E”, “L”, “O”, “W”, “R”, “D”).

For both images, the report will include the results of applying image processing techniques and highlight the importance of pre-processing and hyperparameter tuning in the classification process, discussing the sensitivity of the chosen approach to these changes. Additionally, the classification results on the characters in Image 2 will be reported, following the training of an unsupervised classification model using a provided dataset.

1.2. A Brief Description of the Solution

1.2.1 Solution to Image 1

For Image 1, our initial step involves reading characters from a text file and mapping them onto a 32-level grayscale image based on their ASCII values. In the second step, we perform image binarization using the Otsu thresholding method, transforming the grayscale image into a binary one. This step sets the stage for further image processing. In the third step, we implement image rotation functionality, primarily based on matrix transformations and interpolation techniques. The fourth step involves Sobel operator edge detection, enhancing the detection of object edges in the binary image. The fifth step is the implementation of single-pixel thinning, where we define a thinning function based on iterative principles to reduce the width of objects in the binary image. Lastly, in the sixth step, we introduce a custom function named "label_image" to label different objects and display them in pseudo-colors. This sequence of steps forms the foundation of our approach for Image 1, enabling us to preprocess, enhance, and analyze the image for various applications.

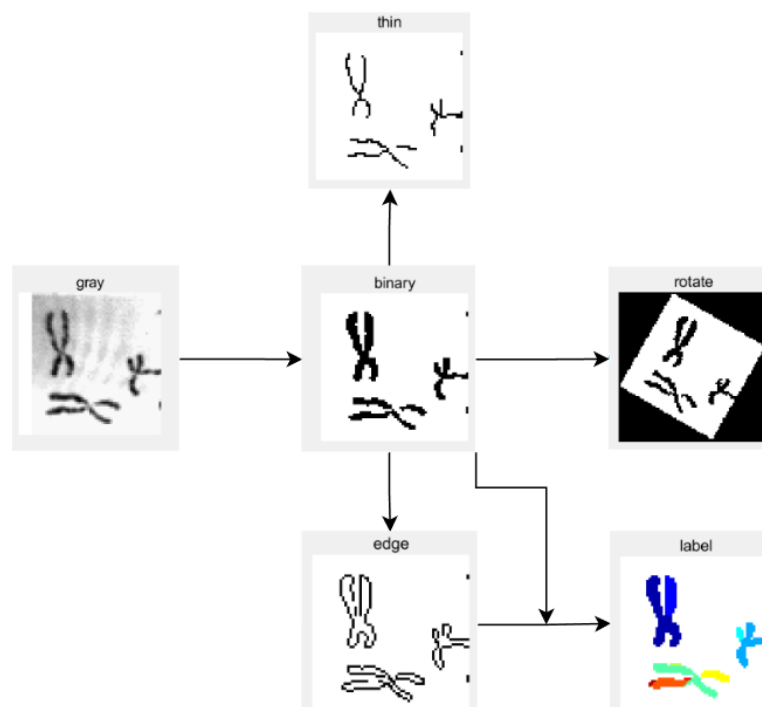


Figure 1-3 Image1 Processing

1.2.1 Solution to Image 2

For the image 2 solution, We first performed a gray image transformation on the color image, which

in turn dovetailed into the processing of the first problem. We performed a series of image processing steps (shown in Figure 1-4), similar to the processing of Image 1, including binarization, edge detection, single-pixel and labeling operations to identify and segment the different letter objects in the image. The final output of our cut letter objects was used as a classification prediction for the machine learning approach. After segmenting the letters from the binary_image, we added multiple columns of white pixels on both sides (shown in Figure 1-5) to fit the main features of the training set to improve accuracy.

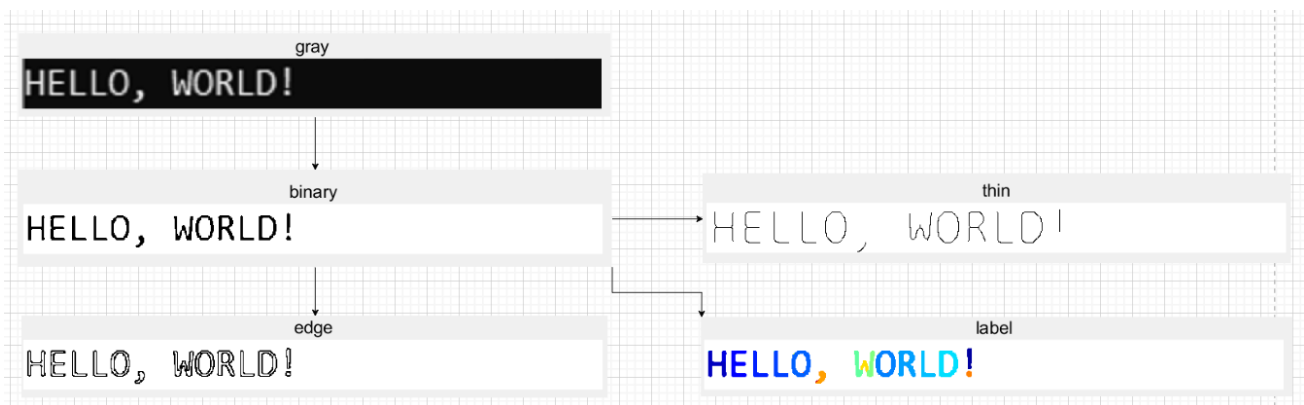


Figure 1-4 Image2 Processing

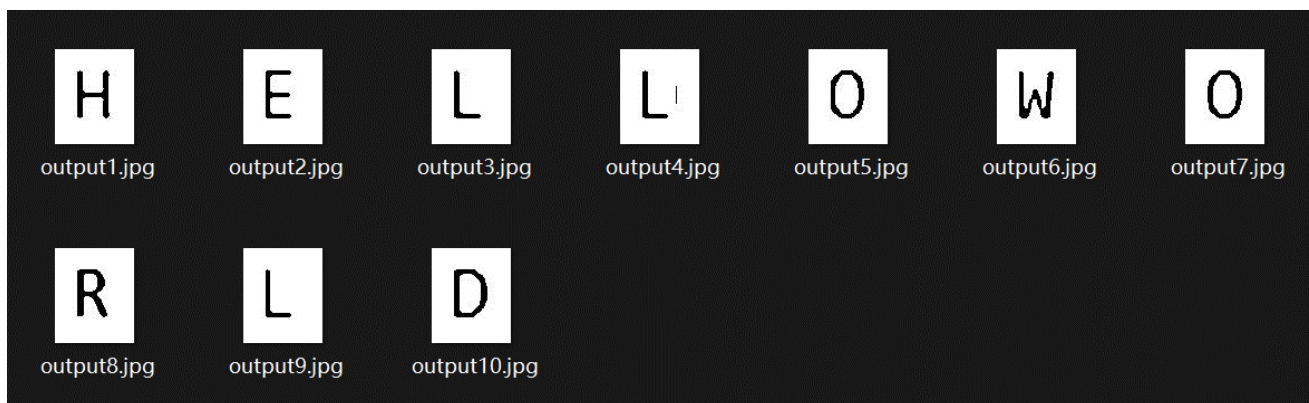


Figure 1-5 Split letters

Most importantly, we used machine learning algorithms, including K-nearest neighbors (KNN), Support vector machine (SVM) and Self-organizing map (Som), to train a model to recognize letters in an image. We use these algorithms to train on the training data set and test on the validation data set to identify letter objects. This gives us a way to automatically recognize characters, thus assigning a label to each character in the image. This comprehensive solution allows us to process the characters in image 2 and split them into separate images so that subsequent ML models can classify them. This scheme not only includes image processing steps, but also involves the application of machine learning

algorithm to improve the accuracy and efficiency of letter recognition.

2. Implementation process

2.1 Binarize

The Otsu method, also known as OTSU or the Maximum Interclass Variance method, is an algorithm used for determining the threshold in image binarization segmentation. This method derives its name from its core principle, which focuses on maximizing the interclass variance between the foreground and background regions in a segmented image. Otsu's method is widely regarded as the optimal choice for threshold selection in image segmentation due to its ability to effectively separate target objects from the background.

The fundamental idea behind Otsu's method is to partition an image into these two regions based on the grayscale values of its pixels. It does so by optimizing the threshold to maximize the interclass variance. When the interclass variance is maximized, it implies that the grayscale distribution of the background and foreground is the most distinct.

One of the key advantages of Otsu's method is its simplicity and computational efficiency. It can determine an optimal global threshold for image binarization without being influenced by variations in image brightness and contrast.

However, like any method, Otsu's method has its limitations. It is sensitive to image noise, which can affect its accuracy. Additionally, it is most effective when segmenting images with a single prominent target. In cases where the size ratio between the target and background regions is significantly imbalanced and the interclass variance function exhibits multiple peaks, the method may not yield optimal results.

In summary, Otsu's method is a powerful tool for image thresholding that excels in many situations, delivering a straightforward and robust approach for image segmentation, particularly when a global threshold is required. Its strengths lie in its simplicity and robustness, making it a valuable asset in the field of digital image processing. Figure 1 shows our overall algorithm flow and pseudocode.


```

Input:
- grayImage: Input grayscale image
Output:
- binaryImage: Binary image

1. Initialize variables and arrays:
- num[256]: Array to store the number of pixels for each grayscale level
- p[256]: Array to store the probability of each grayscale level
- totalmean: Variable to store the total mean gray level of the image
- maxvar: Variable to store the maximum inter-class variance
- point: Variable to store the optimal threshold point

2. Iterate through image pixels to calculate the pixel count and
probability:
FOR i FROM 0 TO 255 DO
    num[i] = CountPixelsWithGrayLevel(grayImage, i)
    p[i] = num[i] / TotalNumberOfPixels(grayImage)
END FOR

3. Calculate the total mean gray level:
totalmean = CalculateTotalMean(p)

4. Initialize maxvar to 0.

5. Iterate through possible thresholds:
FOR k FROM 0 TO 255 DO
    zerosth = CalculateZeroCount(p, k)
    firsth = CalculateFirstMoment(p, k)
    var = CalculateClassVariance(totalmean, zerosth, firsth)
    IF var > maxvar THEN
        maxvar = var
        point = k
    END IF
END FOR

6. Binarize the image using the optimal threshold:
FOR EACH pixel IN grayImage DO
    IF grayImage(pixel) > point THEN
        binaryImage(pixel) = 255 (white)
    ELSE
        binaryImage(pixel) = 0 (black)
    END IF
END FOR

7. Return binaryImage

```

Figure 2-1 OTSU pseudocode

The OTSU algorithm is based on the assumption that there exists a threshold, denoted as "Th," that

can effectively partition all image pixels into two distinct categories: C1 (comprising pixels with values less than Th) and C2 (comprising pixels with values greater than Th). For these two categories, we calculate their respective mean values, denoted as u_0 and u_1 . Simultaneously, we consider the global mean value of the entire image, u . The probabilities of pixels belonging to categories C1 and C2 are represented by w_0 and w_1 , respectively, with the constraint that their sum equals 1.

$$u = w_0 u_0 + w_1 u_1$$

$$w_0 + w_1 = 1$$

In the context of variance, we can express the inter-class variance as follows:

$$\sigma = w_0 \cdot (u_0 - u)^2 + w_1 \cdot (u_1 - u)^2$$

By simplifying the above equation and substituting the given equations, we obtain:

$$\sigma = w_0 \cdot w_1 \cdot (u_0 - u_1)^2$$

During program execution, we iteratively explore the values of Th , ranging from the minimum gray value to the maximum gray value within the image. For each value of T , we aim to maximize the inter-class variance expressed by the formula:

$$\sigma = w_0 \cdot (u_0 - u)^2 + w_1 \cdot (u_1 - u)^2$$

The optimal binarization threshold, denoted as " Th_{best} ," corresponds to the value of Th that yields the maximum inter-class variance:

$$Th_{best} = \max(\sigma)$$

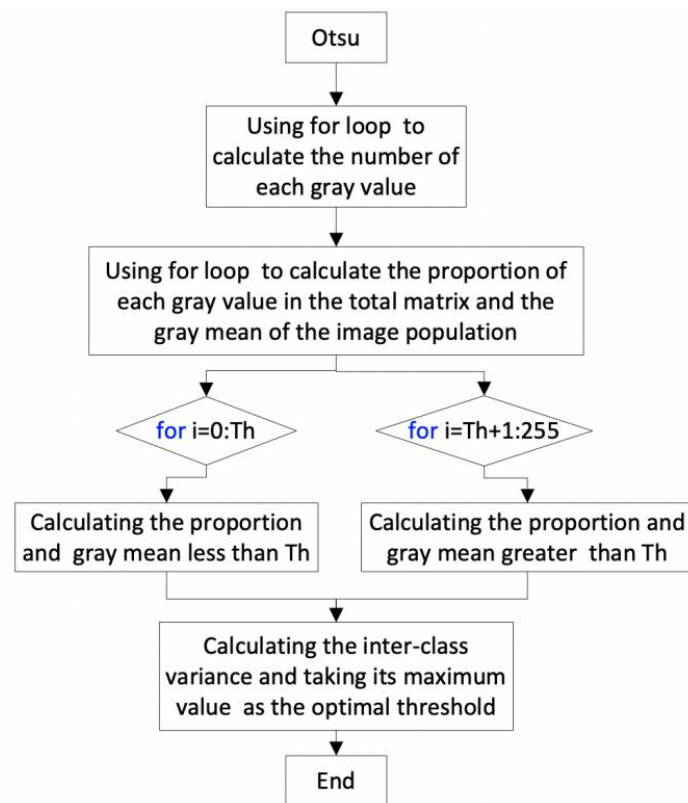


Figure 2-2 OTSU flow

2.2 rotate

```

Function rotateBinaryImage(input: binaryImage, angle):
    angle = convert angle to radians(angle)
    Get dimensions of the binary image (rows, cols)

    Calculate new dimensions for the rotated image (newRows, newCols)

    Initialize a matrix to store the rotated image, size (newRows, newCols)

    Calculate the center coordinates of the rotated image (centerX, centerY)
    Calculate the center coordinates of the original image (originalCenterX, originalCenterY)

    For i from 1 to newRows:
        For j from 1 to newCols:
            Calculate inverse rotation-transformed coordinates x, y

            If x and y are within the original image bounds:
                Calculate bilinear interpolation weights x_fraction, y_fraction

                Fetch surrounding pixel values value1, value2, value3, value4 from the original
image
                Calculate the value of rotatedImage(i, j) using bilinear interpolation

            Binarize rotatedImage: set pixels > 0.5 to 1, <= 0.5 to 0

    Return rotatedImage
  
```

Figure 2-3 Pseudocode of rotate

The overall workflow of the rotation function (Pseudocode in Figure 2-3) is as follows:

1. Determining the Size of the Rotated Image

- It calculates the size of the resulting image after rotation using properties of the rotation matrix, based on the input rotation angle and the original image's dimensions.

2. Creating a New Image to Store the Rotated Result

- It initializes a matrix filled with zeros to store the resulting rotated image.

3. Calculating Centers of Rotation for the Rotated and Original Images

- It computes the coordinates for the center of the rotated image and the center of the original image.

4. Performing Pixel-wise Rotation of the Rotated Image

- It iterates through each pixel in the rotated image, employing an inverse rotation transformation to find its corresponding position in the original image.

- Utilizes bilinear interpolation to calculate the value of the current pixel in the rotated image based on surrounding pixel values in the original image.

5. Binarizing the Rotated Image

- Converts the rotated image to a binary image by setting pixel values above 0.5 to 1 and those less than or equal to 0.5 to 0, producing the final binary image.

Rotation results are shown in Figure 2-4, respectively, 30 degrees 60 degrees and 90 degrees.

This function can be rotated at any angle:

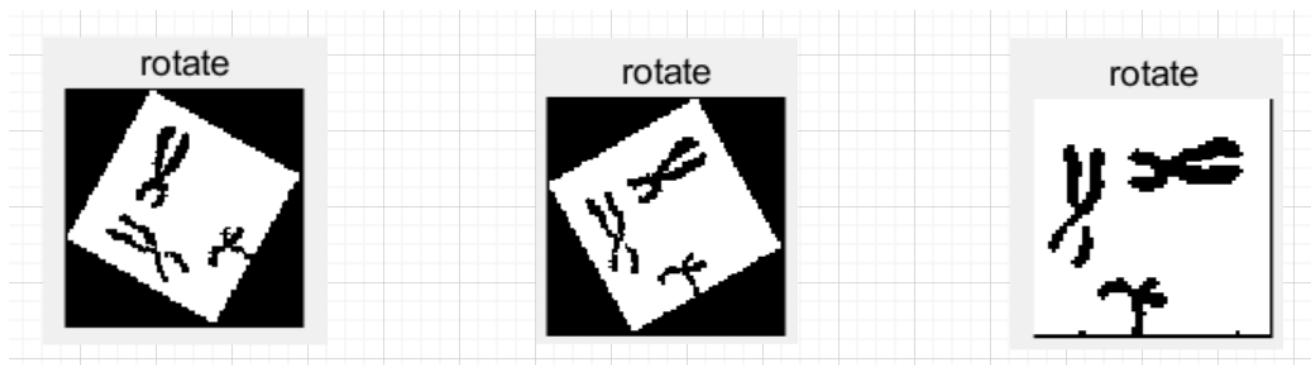


Figure 2-4 Rotate Result

2.3 Edge

```
function edgeImage = SobelEdgeDetection(binaryImage)
    // Get the size of the binary input image
    rows, cols = size(binaryImage)

    // Define the Sobel operators for horizontal and vertical edge detection
    sobelHorizontal = [-1, 0, 1; -2, 0, 2; -1, 0, 1]
    sobelVertical = [-1, -2, -1; 0, 0, 0; 1, 2, 1]

    // Initialize the output edge image with zeros
    edgeImage = zeros(rows, cols)

    // Iterate over each pixel in the binary image
    for i = 2 to rows - 1
        for j = 2 to cols - 1
            // Calculate horizontal gradient using the Sobel operator
            gx = sum(sum(binaryImage[i-1:i+1, j-1:j+1] .* sobelHorizontal))

            // Calculate vertical gradient using the Sobel operator
            gy = sum(sum(binaryImage[i-1:i+1, j-1:j+1] .* sobelVertical))

            // Compute the gradient magnitude
            edgeImage[i, j] = sqrt(gx^2 + gy^2)

        // Normalize the edge image to the range [0, 1]
        edgeImage = edgeImage / max(edgeImage)

    return edgeImage
```

Figure 2-5 sobel edge pseudocode

The algorithm presented in the simplified pseudocode performs edge detection using the Sobel operator on a binary input image. Here is an overview of the algorithm's principles:

1. Input: The algorithm takes a binary image `binary_A` as input, which typically represents an object on a background.
2. Image Size: The size of the binary input image is determined to get the number of rows and columns.
3. Sobel Operator: Two 3x3 Sobel operators are defined for horizontal and vertical edge detection. These operators are used to calculate gradient values for each pixel in the image.

4. Initialization: An output image 'edge_A' is initialized with zeros, which will store the edge information.
5. Convolution: The algorithm performs convolution for horizontal and vertical edge detection using the Sobel operators. It processes each pixel in the binary image, starting from the second row and the second column, up to the second-to-last row and second-to-last column.
6. Gradient Calculation: For each pixel, the horizontal gradient ('gx') and vertical gradient ('gy') are calculated by convolving the Sobel operators with the corresponding neighborhood of the pixel.
7. Gradient Magnitude: The gradient magnitude is computed as the square root of the sum of the squares of 'gx' and 'gy'. This represents the strength of the edge at the current pixel.
8. Edge Image: The computed gradient magnitude is stored in the 'edge_A' image, which represents the edges of the objects in the input image.
9. Normalization: To enhance visualization, the edge image is normalized to the range [0, 1], ensuring that the maximum edge strength corresponds to 1.

In summary, the algorithm uses the Sobel (Figure 2-5) operators to calculate the gradient at each pixel, providing information about the strength and orientation of edges in the binary input image. The output is an edge image that highlights the edges and transitions within the objects in the input image.

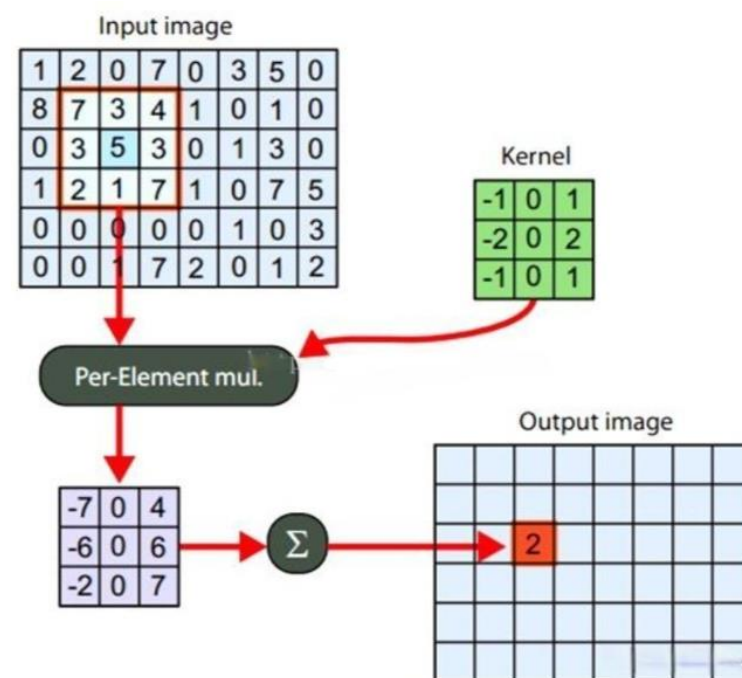


Figure 2-6 Sobel processing

2.4 Thin

```

Function Thin(Binary_A):
    Initialize Thin_A as a copy of the inverted Binary_A
    Set modified flag to true

    While modified is true:
        Set modified flag to false

        For each pixel (i, j) in Binary_A, excluding a 5-pixel border:
            If Binary_A(i, j) is foreground:
                If thin_1(i, j) and thin_2(i, j) and thin_3(i, j) and thin_4(i, j):
                    Mark Thin_A(i, j) for removal
                    Set modified flag to true

    Return Thin_A (the thinned binary image)

Function thin_1(i, j):
    Compute p_round, an array of values in the 8-neighborhood of (i, j)
    If the sum of p_round is between 2 and 6, return true
    Else, return false

Function thin_2(i, j):
    Count the number of transitions from 0 to 1 in the 8-neighborhood
    If there is exactly one transition, return true
    Else, return false

Function thin_3(i, j):
    Check if there is at least one white pixel in the left, top, or right neighbors
    If true, return true
    Else, check if thin_2(i - 1, j) is true, if yes, return true
    Else, return false

Function thin_4(i, j):
    Check if there is at least one white pixel in the top, left, or bottom neighbors
    If true, return true
    Else, check if thin_2(i, j - 1) is true, if yes, return true
    Else, return false
  
```

Figure 2-7 pseudocode of thin function

The 'thin' function shown in Figure 2-7 is the central component responsible for one-pixel thinning of a binary image. It utilizes a set of auxiliary functions, namely 'thin_1', 'thin_2', 'thin_3', and 'thin_4', to decide which pixels should be removed during the thinning process. These auxiliary functions serve specific purposes in the evaluation of pixels, providing criteria to ensure that the thinning operation maintains the object's topology while reducing it to a one-pixel-wide skeleton.

Here's how they work together: The `'thin'` function initiates the process by setting up the `'thin_A'` binary image and flags for iteration. It then enters a loop where it systematically evaluates each pixel of the binary image using the conditions provided by `'thin_1'`, `'thin_2'`, `'thin_3'`, and `'thin_4'`. If a pixel satisfies all of these conditions, it is marked for removal in the intermediate image `'K'`. The process continues iteratively, with the `'modified'` flag determining whether further iterations are necessary. The loop goes on until no more changes (pixel removals) occur. Moreover, here's an explanation of the functions `thin_1`, `thin_2`, `thin_3`, and `thin_4` and how they work in the iterative thinning process:

1. `thin_1`:

It checks if a pixel at coordinates `'(i, j)'` in the binary image `'J'` satisfies the condition of having between 2 and 6 white (foreground) pixels in its 8-neighborhood. The function computes an array `'p_round'` that records the values in the 8-neighborhood of the pixel `'(i, j)'`. It then counts the number of white pixels in this neighborhood. If the count falls within the range of 2 to 6, the function returns `'true'`, indicating that the condition is met. Otherwise, it returns `'false'`.

2. `thin_2`:

It checks if a pixel at coordinates `'(i, j)'` in the binary image `'J'` satisfies the condition of having a single white pixel transition in its 8-neighborhood. This function counts the number of transitions from black (background) to white (foreground) in the 8-neighborhood of the pixel `'(i, j)'`. If there is exactly one transition, it returns `'true'`, indicating that the condition is met. If there are zero or multiple transitions, it returns `'false'`.

3. `thin_3`:

It checks if a pixel at coordinates `'(i, j)'` in the binary image `'J'` satisfies the condition that at least one of its left, top, or right neighbors is white. If not, it also checks if the pixel's top neighbor satisfies `'thin_2'`. The function examines the left, top, and right neighbors of the pixel `'(i, j)'` in the binary image. If at least one of these neighbors is white (foreground), the function returns `'true'`. If not, it checks if the top neighbor `(i - 1, j)` satisfies the `'thin_2'` condition. If any of these conditions is met, it returns `'true'`. Otherwise, it returns `'false'`.

4. `thin_4`:

It checks if a pixel at coordinates `'(i, j)'` in the binary image `'J'` satisfies the condition that at least one

of its top, left, or bottom neighbors is white. If not, it also checks if the pixel's left neighbor satisfies `'thin_2'`. Similar to `'thin_3'`, this function examines the top, left, and bottom neighbors of the pixel `'(i, j)'`. If at least one of these neighbors is white, it returns `'true'`. If not, it checks if the left neighbor `(i, j - 1)` satisfies the `'thin_2'` condition. If any of these conditions is met, it returns `'true'`. Otherwise, it returns `'false'`.

The iterative thinning process uses these functions to evaluate each pixel in the binary image and remove pixels that meet specific thinning criteria. This process continues until no more pixels can be removed (i.e., no further modifications occur, indicated by the `'modified'` flag). The goal is to iteratively thin the binary image while preserving the object's topology and connectivity. These thinning conditions ensure that the object's skeleton remains one-pixel wide.

2.5 Label

```

function label_image(bw, conn = 8)
    rows, cols = size(bw)
    L = zeros(rows, cols)
    next_label = 1

    for i from 1 to rows
        for j from 1 to cols
            if bw(i, j) == 1
                neighbors = find_neighbors(L, i, j, conn)

                if is_empty(neighbors)
                    L(i, j) = next_label
                    next_label = next_label + 1
                else
                    L(i, j) = min(neighbors)
                end
            end
        end
    end

    num = max(L)
end

function find_neighbors(L, i, j, conn)
    rows, cols = size(L)
    neighbors = []

    if conn == 4
        offsets = [[-1, 0], [0, -1]]
    else
        offsets = [[-1, -1], [-1, 0], [-1, 1], [0, -1]]
    end

    for k from 1 to size(offsets)
        x = i + offsets[k, 1]
        y = j + offsets[k, 2]

        if is_valid_pixel(x, y, rows, cols)
            neighbor_label = L(x, y)
            if neighbor_label > 0
                add_to_neighbors(neighbors, neighbor_label)
            end
        end
    end

    return unique(neighbors)
end

function is_valid_pixel(x, y, rows, cols)
    return x >= 1 and x <= rows and y >= 1 and y <= cols
end

function add_to_neighbors(neighbors, label)
    neighbors.append(label)
end

```

Figure 2-8 pseudocode of label_image

The `'label_image'` function shown in Figure 2-8 is designed to label connected components in a binary image (`'bw'`). It employs a connected-component labeling algorithm to assign a unique label to each connected region or object in the input binary image. The `'conn'` parameter allows for the selection of either 4-connectivity or 8-connectivity, which determines how neighboring pixels are considered.

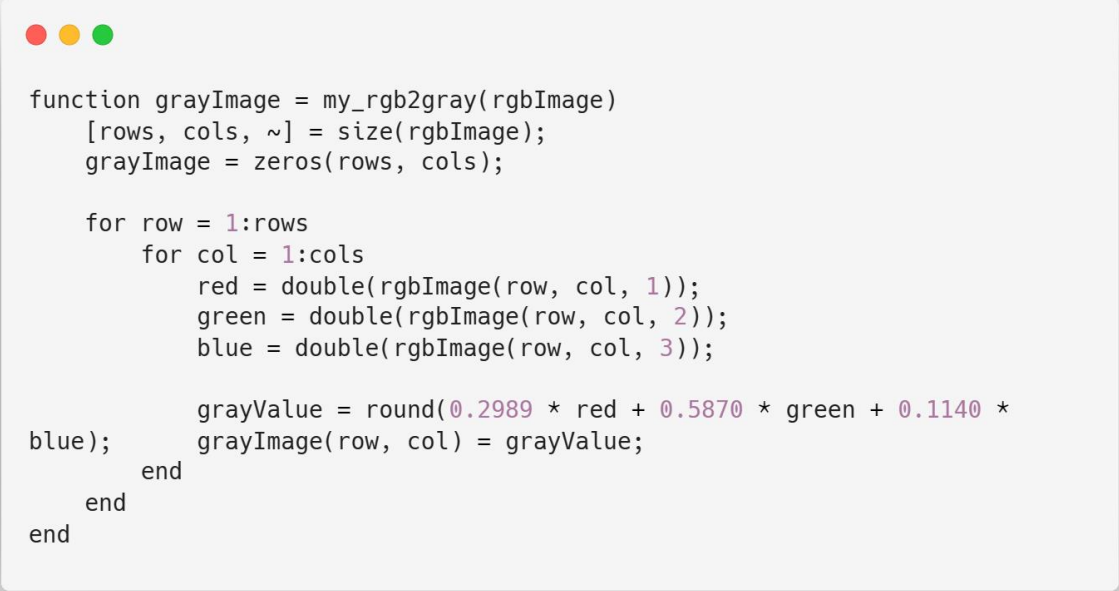
Here's a breakdown of how the `'label_image'` function works:

1. The function initializes an output label matrix `'L'` of the same size as the input binary image. It also initializes a `'next_label'` variable to keep track of the next available label.
2. It checks whether the `'conn'` argument was provided. If not, it defaults to 8-connectivity, which is the most common choice.
3. The function iterates through each pixel of the binary image. For each foreground pixel (pixel value is 1), it examines its neighbors using the `'find_neighbors'` function. This step helps identify neighboring labels.
4. If the current pixel has no labeled neighbors (i.e., it is not part of an already labeled object), the `'next_label'` is assigned to it, and `'next_label'` is incremented to ensure the next object gets a unique label.
5. If the current pixel has labeled neighbors, it assigns the smallest neighboring label to it. This is a key step in maintaining the integrity of connected components. The function avoids giving the same object multiple labels, and connected regions receive the same label.
6. Once all pixels have been processed, the function calculates and returns the number of unique labels found in the image. This value indicates the total number of connected components or objects in the binary image.

The `'find_neighbors'` function is used to identify the labels of neighboring pixels. It examines neighboring pixels according to the chosen connectivity (4-connectivity or 8-connectivity) and returns the unique neighboring labels it finds.

In summary, the `'label_image'` function effectively labels connected components in the binary image, ensuring that each object receives a unique label. This is essential for further analysis and manipulation of objects within the image.

2.6 RgbToGray



```
function grayImage = my_rgb2gray(rgbImage)
    [rows, cols, ~] = size(rgbImage);
    grayImage = zeros(rows, cols);

    for row = 1:rows
        for col = 1:cols
            red = double(rgbImage(row, col, 1));
            green = double(rgbImage(row, col, 2));
            blue = double(rgbImage(row, col, 3));

            grayValue = round(0.2989 * red + 0.5870 * green + 0.1140 *
blue);
            grayImage(row, col) = grayValue;
        end
    end
end
```

Figure 2-9

The RGB to Gray conversion code, as shown in Figure 2-9, is a simple yet fundamental image processing operation. This code demonstrates the process of transforming a color image in RGB format into a grayscale image. The core principle behind this conversion is the weighted average of the red, green, and blue color channels of each pixel. By assigning specific weights to these channels, we can create a grayscale representation that preserves the luminance information while discarding color information. The code iterates through each pixel in the RGB image, calculates the weighted sum of color channels, rounds the result to the nearest integer to obtain the gray value, and builds the grayscale image pixel by pixel. This method is widely used in image processing and computer vision to simplify image analysis and reduce computational complexity while retaining essential luminance data.

3. Machine Learning Methods

3.1 Supervised Classification

In this subsection, we choose multiple supervised learning methods: svm, knn, tree to train the dataset LumiNUS, meanwhile, we have obtained segmented images of different alphabets, and we use them as a test set as a way to test the training effect of the training set in this section.

3.1.1 Feature Extraction

Before proceeding with the machine learning method, we first need to select a certain feature extraction method to extract certain image features from a given image to help the machine learning method for classification. In this part, the HOG (Histogram of Oriented Gradients) method is selected to extract image features, and its main process is as follows:

- 1、 Image Preprocessing: Initially, the image is converted into a grayscale image, typically to simplify calculations and reduce data.
- 2、 Local Gradient Computation: For each pixel in the image, the magnitude and direction of its gradient are calculated. Usually, operators like Sobel are used to compute the horizontal and vertical gradients.
- 3、 Image Block Division: The image is divided into smaller regions (known as cells), and within each region, the gradients' directions of the pixels are statistically analyzed.
- 4、 Orientation Histograms: For each region in which a pixel lies, a histogram is created to show the distribution of gradient directions in that region.
- 5、 Block Normalization: The histograms of adjacent regions are normalized to reduce the impact of lighting variations.

The code is as follows:

```
%% part 1: load file  
path_train = 'G:\zhuomian\5405\assignment\p_dataset_26';
```

```
imds_train = imageDatastore(path_train,'IncludeSubfolders',true,'FileExtensions','.png',...  
'LabelSource','foldernames');
```

```
path_train = 'G:\zhuomian\5405\assignment\ME5405_Group1\test2';  
imds_test = imageDatastore(path_train,'IncludeSubfolders',true,'FileExtensions','.png',...  
'LabelSource','foldernames');
```

```
Train_disp = countEachLabel(imds_train);  
disp(Train_disp);
```

```
%% part2: set paramaters of Hog
```

```
img = readimage(imds_train, 1);
```

```
% Extract HOG features and HOG visualization
```

```
[hog_2x2, vis2x2] = extractHOGFeatures(img,'CellSize',[2 2]);
```

```
[hog_4x4, vis4x4] = extractHOGFeatures(img,'CellSize',[4 4]);
```

```
[hog_8x8, vis8x8] = extractHOGFeatures(img,'CellSize',[8 8]);
```

```
% Show the original image
```

```
figure;
```

```
subplot(2,3,1:3); imshow(img);
```

```
% Visualize the HOG features
```

```
subplot(2,3,4);
```

```
plot(vis2x2);
```

```
title({'CellSize = [2 2]'; ['Length = ' num2str(length(hog_2x2))]});
```

```
subplot(2,3,5);
```

```
plot(vis4x4);
```

```
title({'CellSize = [4 4]'; ['Length = ' num2str(length(hog_4x4))]}));

subplot(2,3,6);
plot(vis8x8);
title({'CellSize = [8 8]'; ['Length = ' num2str(length(hog_8x8))]}));

cellSize = [8 8];
hogFeatureSize = length(hog_8x8);

%% part 3: Get features
Get features of training dataset
numImages = numel(imds_train.Files);
trainingFeatures = zeros(numImages, hogFeatureSize, 'single');

for i = 1:numImages
    img = readimage(imds_train, i);
    img = im2gray(img);

    % Apply pre-processing steps
    img = imbinarize(img);
    trainingFeatures(i, :) = extractHOGFeatures(img, 'CellSize', cellSize);
end

% Get labels for each image.
trainingLabels = imds_train.Labels;

%% Get features of test pictures
numImages = numel(imds_test.Files);
testFeatures = zeros(numImages, hogFeatureSize, 'single');
```



```

for i = 1:numImages
img = readimage(imds_test,i);
img = im2gray(img);
% Apply pre-processing steps
img = imbinarize(img);
testFeatures(i, :) = extractHOGFeatures(img,'CellSize',cellSize);
end
testLabels = imds_test.Labels;

```

In part one, we use the `imageDatastore` function to read the LumiNUS dataset and the test set data obtained in 2.6. After that, in part two, we directly call the HOG library for feature extraction, and set multiple HOG extraction parameters: 2×2 , 4×4 , 8×8 , respectively, to obtain different feature parameters. In this experiment, the feature visualisation obtained by extracting a training set image with different parameters is shown below:

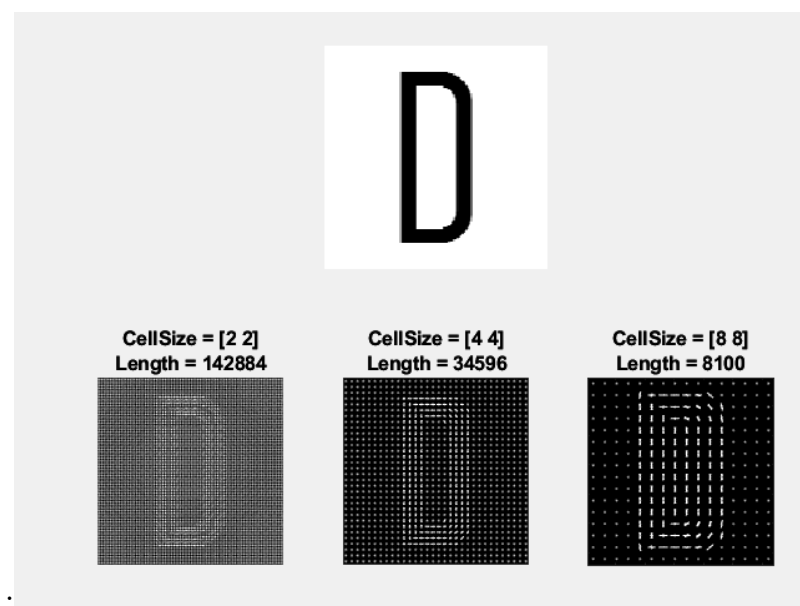


Figure 3-1, Visualization of the HOG features

Among them, the use of 8×8 cell features for 8100 is more appropriate, and too much feature data is extracted when using 4×4 as well as 2×2 , which grows the training time a lot. Therefore, it was finally decided to use 8×8 cell as the HOG parameter for this experiment.

3.1.2 Training Method

For the choice of supervised learning methods, we chose SVM, KNN and Tree.

a) SVM: Support Vector Machine is a supervised learning algorithm commonly used for classification and regression analysis. The main idea is to find an optimal decision boundary (or hyperplane) that can separate data points of different categories.

In classification problems, SVM tries to find a decision boundary that maximises the distance (interval) between two different categories of data points. This interval is determined by the support vector (the data points that support the decision boundary), hence the name SVM.

Advantages of SVM include:

1. Effective in high-dimensional spaces, suitable for cases with high feature dimensionality.
2. Suitable for small sample datasets.
3. By choosing a suitable kernel function, SVM can handle nonlinear problems.

b) KNN: K-Nearest Neighbors is a basic supervised learning algorithm for classification and regression. The principle of KNN is very simple and intuitive: for a new data point, the class of the data point is determined based on the classes of the K known data points closest to it.

Advantages of KNN include:

1. Simple to implement, easy to understand and implement.
2. Works well for multi-category problems and non-linear data.
3. Suitable for small data sets.

c) Tree: decision tree is a commonly used supervised learning algorithm for classification and regression problems. It makes decisions by building a tree structure. In classification problems, each internal node represents a test for a feature, each branch represents a possible value for the test result,

and each leaf node represents a category label. In a regression problem, the leaf nodes represent predicted values for the target variable.

Advantages of Decision Trees:

1. Easy to understand and interpret, good visualisation.
2. Can handle both numerical and categorical data.
3. Less demanding in terms of data preprocessing.

The code for the SVM algorithm is shown below, the KNN and Tree are the same format:

```
%% svm

[trainedClassifier, validationAccuracy] = svm_train(trainingFeatures, trainingLabels);

%
predictedLabels = trainedClassifier.predictFcn(testFeatures)
```

Where svm_train is the training code for the machine learning toolkit, the approximate code with some labels omitted is shown below:

```
function [trainedClassifier, validationAccuracy] = svm_train(trainingData, responseData)
% Extract predictor variables and responses
% The following code processes the data into a suitable shape to train the model.
%
% Convert the input to a table
inputTable = array2table(trainingData, 'VariableNames', {'column_1', 'column_2', 'column_3',
'column_4', 'column_5', 'column_6', 'column_7', ..... 'column_8093', 'column_8094', 'column_8095',
'column_8096', 'column_8097', 'column_8098', 'column_8099', 'column_8100'});

predictorNames = {'column_1', 'column_2', 'column_3', 'column_4', 'column_5', 'column_6',
'column_7', 'column_8', 'column_9', 'column_10', 'column_11', ..... 'column_8087', 'column_8088',
```

```
'column_8089', 'column_8090', 'column_8091', 'column_8092', 'column_8093', 'column_8094',  
'column_8095', 'column_8096', 'column_8097', 'column_8098', 'column_8099', 'column_8100'};  
predictors = inputTable(:, predictorNames);  
response = responseData;  
isCategoricalPredictor = [false, false, false, false, false, false, false, false, .....false, false, false,  
false, false, false, false, false, false, false, false];
```

% Train the classifier

% The following code specifies all the classifier options and trains the classifier.

```
template = templateSVM(...  
'KernelFunction', 'polynomial', ...  
'PolynomialOrder', 2, ...  
'KernelScale', 'auto', ...  
'BoxConstraint', 1, ...  
'Standardize', true);  
classificationSVM = fitcecoc(...  
predictors, ...  
response, ...  
'Learners', template, ...  
'Coding', 'onevsone', ...  
'ClassNames', categorical({'SampleD'; 'SampleE'; 'SampleH'; 'SampleL'; 'SampleO'; 'SampleR';  
'SampleW'}));
```

% Create a result structure using the prediction function

```
predictorExtractionFcn = @(x) array2table(x, 'VariableNames', predictorNames);  
svmPredictFcn = @(x) predict(classificationSVM, x);  
trainedClassifier.predictFcn = @(x) svmPredictFcn(predictorExtractionFcn(x));
```

% Extract predictor variables and responses

% Convert inputs to tables

```
inputTable = array2table(trainingData, 'VariableNames', {'column_1', 'column_2', 'column_3',  
'column_4', 'column_5', 'column_6', 'column_7', 'column_8', ....., 'column_8097', 'column_8098',  
'column_8099', 'column_8100'});
```

```
predictorNames = {'column_1', 'column_2', 'column_3', 'column_4', 'column_5', 'column_6',  
'column_7', 'column_8', 'column_9', 'column_10', ....., 'column_8096', 'column_8097',  
'column_8098', 'column_8099', 'column_8100'};
```

```
predictors = inputTable(:, predictorNames);
```

```
response = responseData;
```

```
isCategoricalPredictor = [false, false, false, false, false, false, false, false, ....., false, false, false,  
false];
```

% Perform cross validation

```
partitionedModel = crossval(trainedClassifier.ClassificationSVM, 'KFold', 25);
```

% Computational validation of forecasts

```
[validationPredictions, validationScores] = kfoldPredict(partitionedModel);
```

% Calculated validation accuracy

```
validationAccuracy = 1 - kfoldLoss(partitionedModel, 'LossFun', 'ClassifError');
```

This function trains the model parameters with the training set and returns the validation accuracy validationAccuracy and the trained model trainedClassifier. Afterwards, the predictFcn(testFeatures) function is used to obtain the results of the prediction of the test set labels using the model. The prediction results of the labels of the test set using this model can be obtained by using the function trainedClassifier. In this paper, we directly use Classification Learner Apps of matlab for training and calculate the confusion matrix during training. After that the prediction is done by the model.

3.1.3 Results

In this experiment, 75% of the dataset is used as the training set and 25% as the validation set for all three machine learning methods.

a) The training and prediction results of a, SVM are shown below:

The accuracy for the validation set is shown below, 99.59%.

```
validationAccuracy =  
  
single  
  
0.9959
```

Figure 3-2: Validation Accuracy of SVM

Its confusion matrix is shown below and it can be found that the verification using SVM is great.

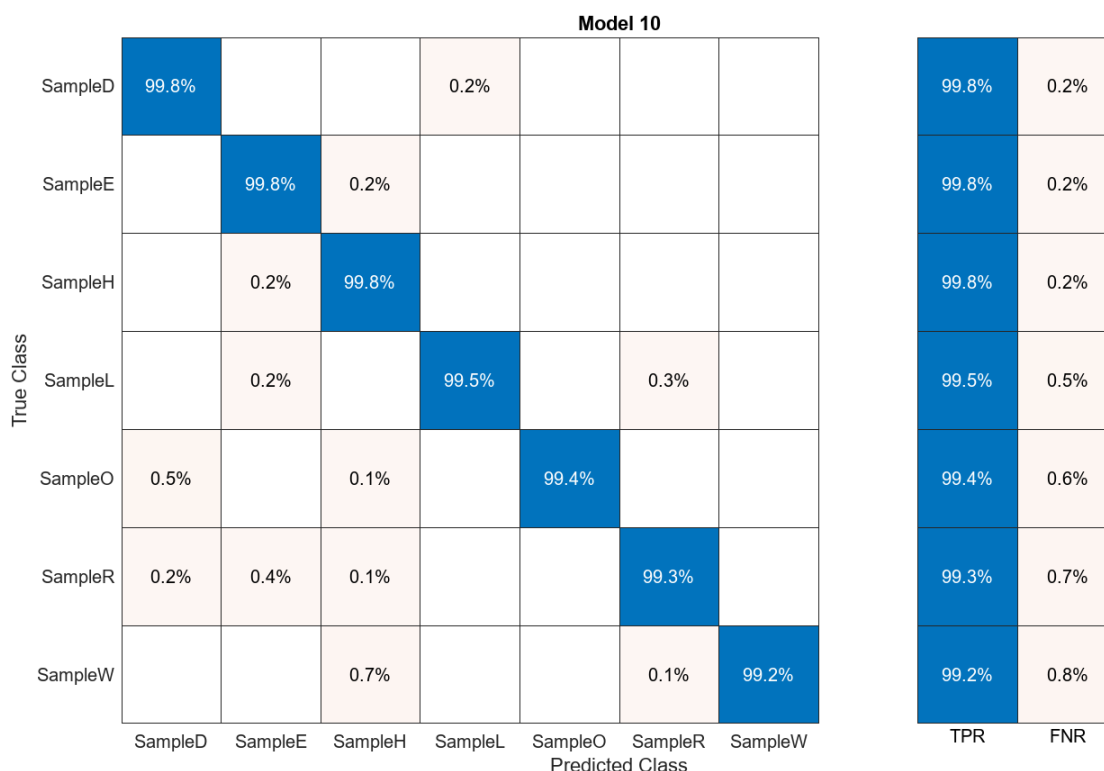


Figure 3-3 : Confusion Matrix of Svm

Prediction of segmented letters is done using SVM:

	1	2
1 D		
2 E		
3 H		
4 L		
5 O		
6 R		
7 W		
8		

Figure 3-4 : Prediction of SVM

b) Cosine KNN

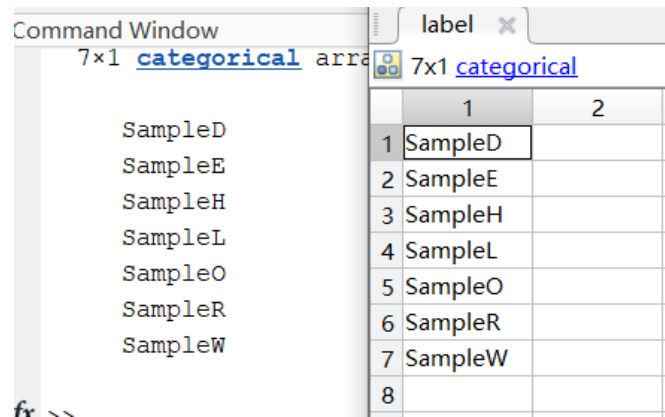
validationAccuracy =
single
0.9889

Figure 3-5 : Validation Accuracy of KNN

		Model 6.4								
True Class	SampleD	96.5%	0.2%	0.3%	0.6%	1.8%	0.4%	0.3%	96.5%	3.5%
	SampleE	0.6%	93.3%	1.3%	1.7%	2.3%	0.5%	0.4%	93.3%	6.7%
	SampleH	0.5%	0.7%	96.4%	0.1%	0.1%	0.7%	1.6%	96.4%	3.6%
	SampleL	0.5%	1.2%	1.1%	96.0%	0.7%	0.6%		96.0%	4.0%
	SampleO	0.5%		0.2%		99.3%			99.3%	0.7%
	SampleR	1.3%	0.5%	1.8%	0.3%		96.1%	0.1%	96.1%	3.9%
	SampleW	0.4%	0.1%	2.4%		0.6%	0.3%	96.3%	96.3%	3.7%
		SampleD	SampleE	SampleH	SampleL	SampleO	SampleR	SampleW	TPR	FNR

Figure 3-6 : Confusion Matrix of KNN

As can be seen from the validation accuracy as well as the confusion matrix, the KNN works well on the training set. On the test set, the results are as follows:



	1	2
1 SampleD		
2 SampleE		
3 SampleH		
4 SampleL		
5 SampleO		
6 SampleR		
7 SampleW		
8		

Figure 3-7: Prediction of KNN

Predicted all 7 letters correctly with great results.

c) Fine Tree:

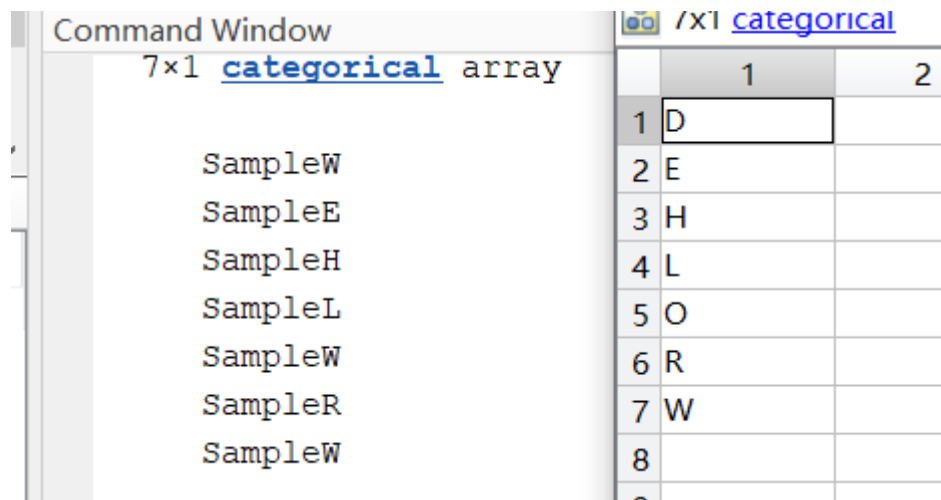
```
validationAccuracy =  
  
single  
  
0.9183
```

Figure 3-8 : Validation Accuracy of Tree

		Model 2.1								
True Class	SampleD	92.4%	0.5%	1.0%	1.7%	1.3%	0.7%	2.5%	92.4%	7.6%
	SampleE	1.1%	91.1%	3.5%	0.6%	0.8%	2.4%	0.5%	91.1%	8.9%
	SampleH	1.4%	3.0%	91.2%	0.7%	0.6%	1.7%	1.5%	91.2%	8.8%
	SampleL	0.9%	0.9%	0.8%	94.5%	0.7%	1.0%	1.3%	94.5%	5.5%
	SampleO	4.1%	0.4%	1.6%	0.5%	92.1%	0.6%	0.7%	92.1%	7.9%
	SampleR	2.0%	3.1%	3.2%	1.5%	0.1%	88.8%	1.3%	88.8%	11.2%
	SampleW	2.1%	0.9%	2.5%	0.7%	0.9%	1.3%	91.7%	91.7%	8.3%
		SampleD	SampleE	SampleH	SampleL	SampleO	SampleR	SampleW	TPR	FNR
		Predicted Class								

Figure 3-9: Confusion Matrix of Tree

The validation set accuracy when using Fine Tree is reduced compared to KNN as well as SVM algorithms, but there are also good results.



Command Window

7x1 categorical array

	1	2
1 D		
2 E		
3 H		
4 L		
5 O		
6 R		
7 W		
8		

SampleW
SampleE
SampleH
SampleL
SampleW
SampleR
SampleW

Figure 3-10: Prediction of Tree

There were prediction errors in two images, but the overall accuracy was still high.

3.2 Influencing Factors

3.2.1 Hyperparameter

In 3.1, for the selection of KNN models, we tried a number of different KNNs, such as Fine KNN, Medium KNN, and finally Cosine Knn. the first two KNNs were equally effective on the training set, with 98.8% and 95.2% accuracy, respectively, and their confusion matrices are shown in the following figure:

Model 4

SampleD	1004	2		1	4	5	
SampleE		995	3	16		2	
SampleH	1	2	1009	1	1	2	
SampleL	4	6		1006			
SampleO	4				1012		
SampleR	5	2	12	1		996	
SampleW	1		8			2	1005
	SampleD	SampleE	SampleH	SampleL	SampleO	SampleR	SampleW

Predicted Class

Figure 3-11: Confusion Matrix of Fine KNN

Model 6.2

SampleD	983	1	3	11	15	3	
SampleE	11	924	23	22	32	1	3
SampleH	4	11	979	4	3	4	11
SampleL	7	12	12	976	7	2	
SampleO	8		3	1	1004		
SampleR	28	12	28	13	2	932	1
SampleW	8	1	29		3	5	970
	SampleD	SampleE	SampleH	SampleL	SampleO	SampleR	SampleW

Predicted Class

Figure 3-12: Confusion Matrix of Medium KNN

However, the model trained with these two KNN parameters is less effective in recognising the letters segmented in 2.6, as shown below:

```
>> label=fineknnModel.predictFcn(testFeatures)

label =

7×1 categorical array

SampleL
SampleL
SampleL
SampleL
SampleO
SampleL
SampleL
```

Figure 3-13: Prediction of Fine KNN

Most of the letters were predicted to be 'L'. It is evident that adjusting the parameters in the KNN has an impact on the test results.

3.2.2 Padding Factor

Due to the large number of training set images, we chose to keep the training images unchanged while performing the training, while making predictions by adjusting the parameters of the test set letter images. When using the HOG feature extraction method, it can be noticed that since the letters of the training set images are generally located in the centre of the images, the features extracted by HOG have smaller feature data at the edges of the images as shown in their feature map visualisation:

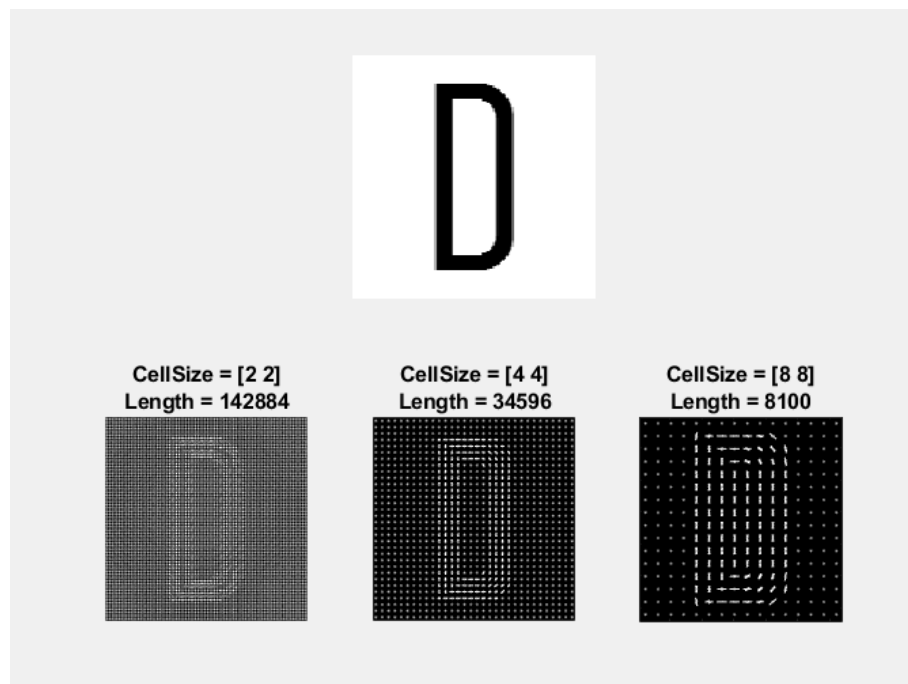


Figure 3-14: Feature Map Visualisation of Data in Center

If the test set image has more useful pixels at the edges, as shown in the figure below, it will result in poor prediction:

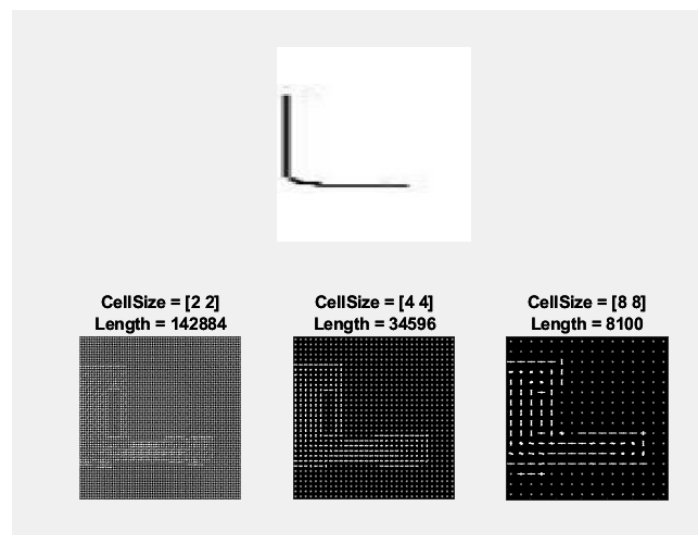


Figure 3-15: Feature Map Visualisation

Therefore, in the final choice, we chose to fill white pixels around the test set, as shown in the figure below, which can effectively concentrate the pixels with the presence of information in the middle and improve the classification accuracy, shown below:

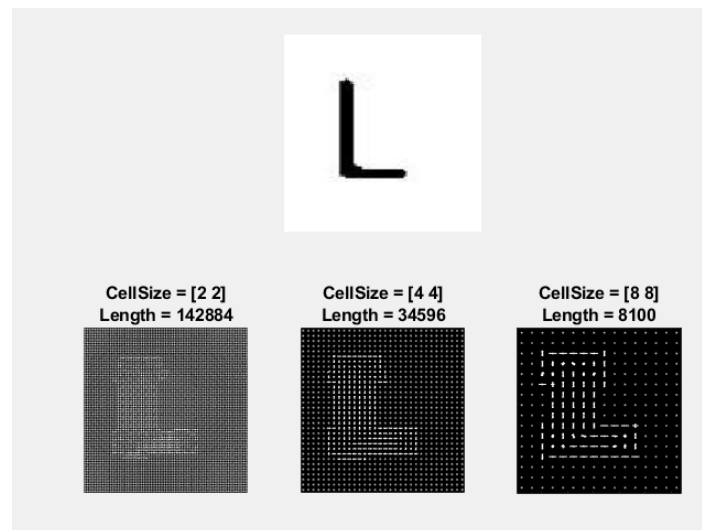


Figure 3-16: Feature Map Visualisation

3.2.3 Pixel Size Factor

The image pixels of the training set are 128×128 and when the pixels of the test set image are modified, there is a large difference in the number of features extracted due to the HOG method used for feature extraction. The number of features when using 128×128 pixel image is 8100 and when using 34×85 the number of features is 974 having different feature dimensions. Due to the large number of training sets, it is necessary to change the dimensionality parameter of the test images while considering the training set images unchanged, otherwise the prediction will not be possible.

3.2.4 Binarization

Before extracting the features, it is first binarised and processed, which can effectively help HOG to extract more effective features, and its feature visualisation diagram is shown below, which can improve the classification accuracy.

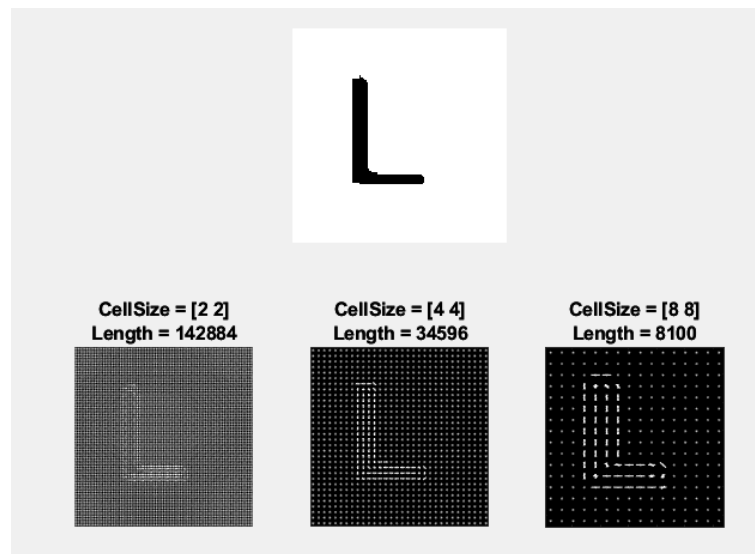


Figure 3-17: Feature Map Visualisation After Binarization

Therefore, in the choice of parameters for machine learning, we chose to use 128*128 pixel images, while filling the test set images, and binarised them first during feature extraction, and chose the Cosine parameter for KNN training to obtain a model with excellent classification results.

4. Appendix

Qs1.m

```
1.
   %Group1 CA Project1
2.
3. clear;
4. close all;
5. %%
6. fid = fopen("ME5405_Group1\chromo.txt");
7.
8. lf = newline;
9. cr = char(13);
10.
11. A = fscanf(fid,[cr lf '%c'],[64,64]);
12. fclose(fid);
13. A = A';
14. %%denoise
15. A(:, 1:6) = 32;
16.
17. A(isletter(A))= A(isletter(A)) - 55;
18. A(A >= '0' & A <= '9') = A(A >= '0' & A <= '9') - 48;
19. gray_A = uint8(A);
20. %%
21. figure;
22. subplot(3,2,1);
23. imshow(gray_A, []);
24. title('gray');
25. hold on;
26.
27. %% Binarize
28. subplot(3,2,2);
29. threshold = otsuThreshold(gray_A);%otsu
30. binary_A = gray_A > threshold;
31. imshow(binary_A, []);
32. title('binary');
33.
34. %% rotate
35. subplot(3,2,3);
36. rotate_A = rotateBinaryImage(binary_A,90);%the second parameter is rotate angle
```

```

37.imshow(rotate_A,[]);
38.title('rotate');
39.
40.%% Edge
41.subplot(3,2,4);
42.edge_A = edge(binary_A, 'sobel');
43.%edge_A = edge_sobel(binary_A);
44.edge_A = ~edge_A;
45.imshow(edge_A,[]);
46.title('edge');
47.%% thin
48.subplot(3,2,5);
49.thin_A = thin(binary_A);
50.imshow(thin_A, []);
51.title('thin');
52.%% label
53.bw = ~binary_A | ~edge_A;
54.% [L,num] = bwlabel(bw,8);
55.[L,num] = label_image(bw,8);
56.RGB = label2rgb(L);
57.subplot(3,2,6),imshow(RGB),title('label')

```

Qs2.m

```

1. clear;
2. close all;
3. %%
4. inputImage = imread('hello_world.jpg');
5. %divide
6. [rows, cols, ~] = size(inputImage);
7. third = floor(rows / 3);
8. subImage = inputImage((third + 1):(2 * third), :, :);
9. imwrite(subImage, 'subImage.jpg');
10.%rgb to gray
11.gray_A = my_rgb2gray(subImage);
12.%%
13.figure;
14.subplot(3,2,1);
15.imshow(gray_A, []);
16.title('gray');
17.hold on;
18.
19.% Binarize
20.subplot(3,2,2);
21.threshold = otsuThreshold(gray_A) + 50;%otsu

```



```
22.binary_A = gray_A < threshold;
23.binary_A(:, 1:6) = 1;%denoise
24.imshow(binary_A,[]);
25.title('binary');
26.
27.% Edge
28.subplot(3,2,3);
29.%edge_A = edge(binary_A, 'sobel');
30.edge_A = edge_sobel(binary_A);
31.edge_A = ~edge_A;
32.imshow(edge_A,[]);
33.title('edge');
34.
35.%thin
36.subplot(3,2,4);
37.thin_A = thin(binary_A);
38.imshow(thin_A, []);
39.title('thin');
40.
41.%label
42.bw = ~binary_A | ~edge_A;
43.% [L,num] = bwlabel(bw,8);
44.[L,num] = label_image(bw,8);
45.RGB = label2rgb(L);
46.subplot(3,2,5),imshow(RGB),title('label')
47.%% segment
48.num_images = 10;
49.output_folder = 'output';
50.seg_A = thin_A;
51.%seg_A = binary_A;
52.
53.if ~exist(output_folder, 'dir')
54.    mkdir(output_folder);
55.end
56.
57.center_row = round(size(seg_A, 1) / 2 - 5);
58.image_count = 0;
59.
60.col = 1;
61.while col <= size(seg_A, 2) && image_count < num_images
62.    if seg_A(center_row, col) == 0
63.        start_col = col - 4;
64.        end_col = min(start_col + 33, size(seg_A, 2));
```

```

65.         cropped_image = seg_A(:, start_col:end_col);
66.
67.         cropped_image = [ones(size(cropped_image, 1), 20) cropped_image]
        ;
68.         cropped_image = [cropped_image ones(size(cropped_image, 1), 20)]
        ;
69.
70.         output_filename = fullfile(output_folder, sprintf('output%d.jpg'
        , image_count + 1));
71.         imwrite(cropped_image, output_filename);
72.
73.         image_count = image_count + 1;
74.
75.         col = end_col + 1;
76.     else
77.         col = col + 1;
78.     end
79. end
80.
81.

```

model.m

```

1. %% svm
2.
3. [trainedClassifier, validationAccuracy] = svm_train(trainingFeatures, tr
    ainingLabels);
4. %
5. % label=svmtestModel.predictFcn(testFeatures)
6. predictedLabels = trainedClassifier.predictFcn(testFeatures)
7.
8. confMat = confusionmat(testLabels, predictedLabels);
9.
10. %% knn
11.
12. [trainedClassifier, validationAccuracy] = knncosineClassifier(trainingFe
    atures, trainingLabels)
13. predictedLabels = trainedClassifier.predictFcn(testFeatures)
14. % label=fineknnModel.predictFcn(testFeatures)
15. % label=cosineModel.predictFcn(testFeatures)
16.
17. %% fine tree
18.
19. [trainedClassifier, validationAccuracy] = tree_train(trainingFeatures, t
    rainingLabels)

```

```
20.predictedLabels = trainedClassifier.predictFcn(testFeatures)
21.% label=treeModel.predictFcn(testFeatures)
```

featureExtraction.m

```
1. %% load file
2. path_train = 'G:\zhuomian\5405\assignment\p_dataset_26';
3. imds_train = imageDatastore(path_train,'IncludeSubfolders',true,'FileExt
   ensions','.png',...
4.                               'LabelSource','foldernames');
5.
6. path_train = 'G:\zhuomian\5405\assignment\ME5405_Group1\output (binary)'
   ;
7. imds_test = imageDatastore(path_train,'IncludeSubfolders',true,'FileExte
   nsions','.jpg',...
8.                               'LabelSource','foldernames');
9.
10.Train_disp = countEachLabel(imds_train);
11.disp(Train_disp);
12.%%
13.img = readimage(imds_train, 1);
14.
15.% Extract HOG features and HOG visualization
16.[hog_2x2, vis2x2] = extractHOGFeatures(img,'CellSize',[2 2]);
17.[hog_4x4, vis4x4] = extractHOGFeatures(img,'CellSize',[4 4]);
18.[hog_8x8, vis8x8] = extractHOGFeatures(img,'CellSize',[8 8]);
19.
20.% Show the original image
21.figure;
22.subplot(2,3,1:3); imshow(img);
23.
24.% Visualize the HOG features
25.subplot(2,3,4);
26.plot(vis2x2);
27.title({'CellSize = [2 2]'; ['Length = ' num2str(length(hog_2x2))]});
28.
29.subplot(2,3,5);
30.plot(vis4x4);
31.title({'CellSize = [4 4]'; ['Length = ' num2str(length(hog_4x4))]});
32.
33.subplot(2,3,6);
34.plot(vis8x8);
35.title({'CellSize = [8 8]'; ['Length = ' num2str(length(hog_8x8))]});
36.
37.cellSize = [8 8];
```

```
38.hogFeatureSize = length(hog_8x8);
39.
40.
41.%%
42.numImages = numel(imds_train.Files);
43.trainingFeatures = zeros(numImages,hogFeatureSize,'single');
44.
45.for i = 1:numImages
46.    img = readimage(imds_train,i);
47.
48.    img = im2gray(img);
49.
50.    % Apply pre-processing steps
51.    img = imbinarize(img);
52.
53.    trainingFeatures(i, :) = extractHOGFeatures(img,'CellSize',cellSize)
    ;
54.end
55.
56.% Get labels for each image.
57.trainingLabels = imds_train.Labels;
58.
59.
60.%%
61.img = readimage(imds_test, 4);
62.    img = im2gray(img);
63.
64.    % Apply pre-processing steps
65.    img = imbinarize(img);
66.% Extract HOG features and HOG visualization
67.[hog_2x2, vis2x2] = extractHOGFeatures(img,'CellSize',[2 2]);
68.[hog_4x4, vis4x4] = extractHOGFeatures(img,'CellSize',[4 4]);
69.[hog_8x8, vis8x8] = extractHOGFeatures(img,'CellSize',[8 8]);
70.
71.% Show the original image
72.figure;
73.subplot(2,3,1:3); imshow(img);
74.
75.% Visualize the HOG features
76.subplot(2,3,4);
77.plot(vis2x2);
78.title({'CellSize = [2 2]'; ['Length = ' num2str(length(hog_2x2))]});
79.
```

```
80.subplot(2,3,5);
81.plot(vis4x4);
82.title({'CellSize = [4 4]'; ['Length = ' num2str(length(hog_4x4))]});
83.
84.subplot(2,3,6);
85.plot(vis8x8);
86.title({'CellSize = [8 8]'; ['Length = ' num2str(length(hog_8x8))]});
87.cellSize = [8 8];
88.hogFeatureSize = length(hog_8x8);
89.%%
90.numImages = numel(imds_test.Files);
91.testFeatures = zeros(numImages,hogFeatureSize,'single');
92.
93.for i = 1:numImages
94.    img = readimage(imds_test,i);
95.
96.    img = im2gray(img);
97.
98.    % Apply pre-processing steps
99.    img = imbinarize(img);
100.
101.    testFeatures(i, :) = extractHOGFeatures(img,'CellSize',cellSize);
102. end
103.    testLabels = imds_test.Labels;
104.
105.
106.
```