**Table of Content**

|  |  |
| --- | --- |
| Content | Page number |
| Chapter 1 Background | 5 |
| Chapter 1.1 Problem Statement and Motivation | 5 |
| Chapter 1.2 Objectives | 5 |
| Chapter 1.3 Project Scope | 5 |
| Chapter 1.4 Contributions | 6 |
|  |  |
| Chapter 3 System Design | 7 |
| Chapter 3.1 Segmentation | 7 |
| Chapter 3.2 Classification | 14 |
|  |  |
| Chapter 4 Experiments and Simulations | 18 |
| Chapter 4.1 Implementation Issues and Challenges during Segmentation | 18 |
| Chapter 4.2 Implementation Issues and Challenges during Classification | 19 |
|  |  |
| Chapter 5 System evaluation and discussion | 20 |
| Chapter 5.1 System Performance Definition | 20 |
| Chapter 5.2 Results | 21 |
|  |  |
| Chapter 6 Conclusion and recommendation | 22 |
| Chapter 6.1 Conclusion | 22 |
| Chapter 6.2 Novelty of work | 22 |
| Chapter 6.3 Future work | 23 |
|  |  |
| Appendix | 24 |

**List of Figures**

|  |  |
| --- | --- |
| Figure | Page number |
| Figure 3.1.1 Segmentation flow 1 | 8 |
| Figure 3.1.2 Colour segmentation result | 9 |
| Figure 3.1.4 Binary and operation between the colour mask and the filled longest edge mask | 10 |
| Figure 3.1.5 Removing very small noises in the image | 11 |
| Figure 3.1.6 Removing large noises in the image | 12 |
| Figure 3.1.7 Segmentation flow 2 | 13 |
| Figure 3.2.1 Vertical and horizontal separations of the images | 15 |
| Figure 3.2.2 Horizontally flipped country flag. | 15 |
| Figure 3.2.3 Two additional middle areas | 16 |
| Figure 4.1.1 Noise reduction strategy cut the flag because of complicated pattern | 18 |
| Figure 4.1.2 Noise reduction strategy cut the flag because of wrinkled flag | 19 |
| Figure 4.2.1 Extremely similar flags | 19 |
| Figure 5.1 1 Intersection over union | 20 |

**List of Tables**

|  |  |
| --- | --- |
| Table | Page number |
| Table 3.1.1 Colour ranges for segmentation flow 1 | 7 |
| Table 3.1.2 Colour ranges for segmentation flow 2 | 9 |
| Table 5.2.1 Segmentation accuracy and classification accuracy | 21 |

**Chapter 1: Background**

**1.1 Problem Statement and Motivation**

Recognizing all flags in this world is difficult for humans. There are more than 200 countries in the world, and many country flags look similar. A flag recognizing system can help people to recognize them whenever they see an unfamiliar flag. For computer systems, recognizing flags is hard because real-world images may contain a lot of noises, such as complicated backgrounds, which may affect the accuracy of segmentation and recognition. Some images may also be captured under low contract, insufficient illumination, and low brightness environment. Besides, the flags captured in the real world may also be wrinkled, rotated, or flipped, making it hard to segment and extract features from them.

**1.2 Objectives**

The main goal of this project is to develop a flag recognition system that can classify the flags given an input image. Thus, the objectives of this project are:

* Able to segment the flags out of their background.
* Able to classify the flag given the segmented region.
* Able to get a high accuracy for both segmentation and classification.
* Able to produce the result on time and should take less than one second to process each image.

**1.3 Project Scope**

A flag recognition system will be developed using C++ programming language with Open-Source Computer Vision Library (OpenCV) on Visio Studio 2019. The system can be divided into two stages: flag segmentation and flag classification. During the segmentation stage, the system will differentiate the flags with their backgrounds and crop them out. In the classification stage, the system will extract the features from the cropped images and feed them into a traditional machine-learning model to get the predicted country.

**1.4 Contributions**

This project will investigate the existing strategies and discover new strategies to segment and classify the country flags. This project will also experiment with different settings of the strategies to find the best settings that could yield the best accuracy. The flag recognition system proposed in this assignment should have improved performance, especially when the flags are wrinkled, rotated, and flipped. The proposed strategies could also be used on other recognition systems that need to recognise similar objects to improve their accuracy.

**Chapter 3: System Design**

**3.1 Segmentation**

Since the images are from two domains, different segmentation flows are used to handle each domain. One of the domains is cropped images from Set A, B, and D, and another is uncropped images from Set C. A very simple HSV colour-based segmentation is used to handle the cropped images domain and a much more complicated segmentation flow for the another.

For the cropped images, the images are first blurred using median blur with kernel size 3x3 to remove noises from the image. Next, HSV colour segmentation is used to segment out the common colours from the flags. The common colours are red, yellow, green, blue, white, and black. The table below are the range of the colours.

Table 3.1.1. Colour ranges for segmentation flow 1.

|  |  |  |
| --- | --- | --- |
| Colour | Lower threshold | Upper threshold |
| Red 1 | (0, 110, 140) | (15, 255, 255) |
| Red 2 | (165, 110, 140) | (180, 255, 255) |
| Yellow | (15, 85, 50) | (40, 255, 255) |
| Green | (50, 100, 100) | (85, 255, 255) |
| Blue | (75, 70, 50) | (140, 255, 255) |
| White | (0, 0, 190) | (180, 80, 255) |
| Black | (0, 0, 0) | (180, 255, 45) |

After segmenting the colours, binary masks containing the pixels segmented from each colour range are combined using bitwise or operation. Then, the morphological close operation is used to fill the small gaps inside the combined binary mask. A rectangular bounding box is then drawn to contain the area with segmented pixels. Since the images are cropped, the bounding box should contain the whole image. The figure below shows the segmentation flow.

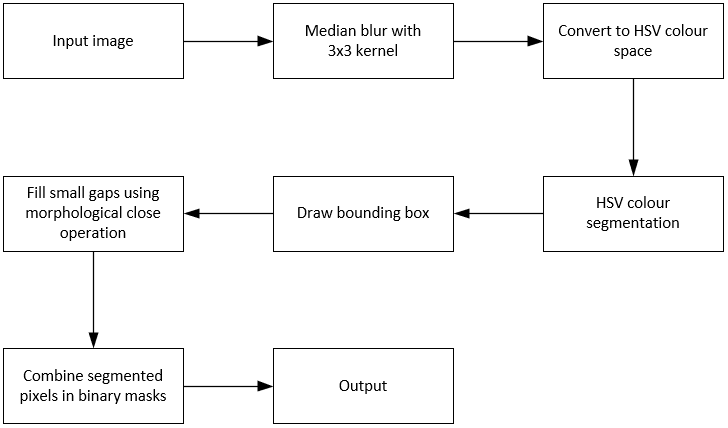


Figure 3.1.1. Segmentation flow 1

For the segmentation of uncropped images, a novel segmentation flow is proposed to remove the background of the images. When the image is fed into the program, the image is first blurred using Gaussian blur with a 3x3 kernel to reduce noises in the image before converting to HSV colour space. One special feature about the country flags to differentiate them from the background is that their colours are highly saturated. Therefore, the pixels containing the flag could be segmented according to their saturation. The mean saturation value of the flag is first calculated, and then the pixels whose saturation value falls in the range between the mean saturation value and the maximum saturation value are segmented and stored in a binary mask. The black and white pixels are also segmented but they are segmented separately because black and white pixels have a very low saturation value. The binary masks containing the segmented pixels are then combined using the bitwise or operation, and small gaps between them are filled using the morphological close operation. Table 3.1.2 shows the range of the colours, and figure 3.1.2 shows some of the results. Note that the ranges are adjusted according to the dataset in set C to get a better result.

Table 3.1.2. Colour ranges for segmentation flow 2.

|  |  |  |
| --- | --- | --- |
| Colour | Lower threshold | Upper threshold |
| More saturated pixels | (0, mean saturation, 0) | (180, 255, 255) |
| Black | (0, 0, 0) | (180, 130, 50) |
| White | (0, 0, 130) | (180, 55, 255) |

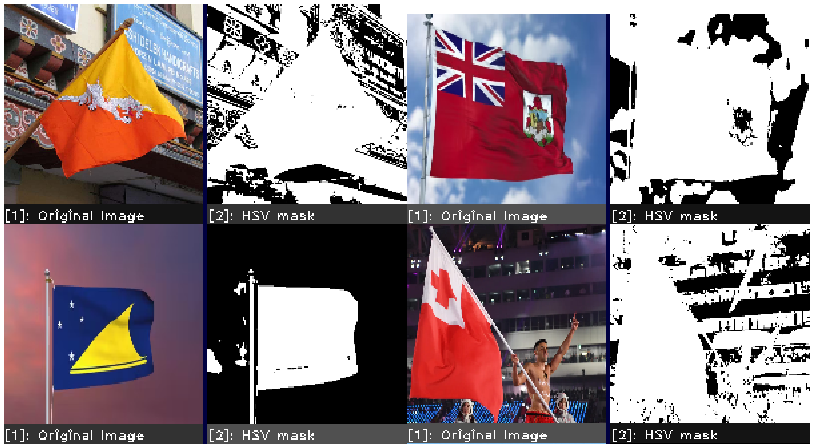


Figure 3.1.2. Colour segmentation result.

Canny edge is also used to extract the edges, including the noises from the background in the flags. The original image is first blurred using Gaussian blur with a 3x3 kernel and converted into HSV colour space before extracting the edges. The blurred HSV image is split into three channels, and canny edge operation is performed separately on the H and S channels. The V channel is ignored because the lighting might add undesired noises. The lower and upper threshold is set to 1 and 200 for both channels to detect very small changes in the image. Then, the two binary masks containing segmented edges from the two channels are combined using the bitwise or operation. The longest contour in the combined mask is found, filled, and drawn on a new mask. After getting the masks containing the more saturated colours and the longest edge, the binary and operation is performed between the two masks. The intention of this step is to remove any the undesired pixels segmented that is not part of the flag in the colour mask. Figure 3.1.3 shows some of the results after these steps.

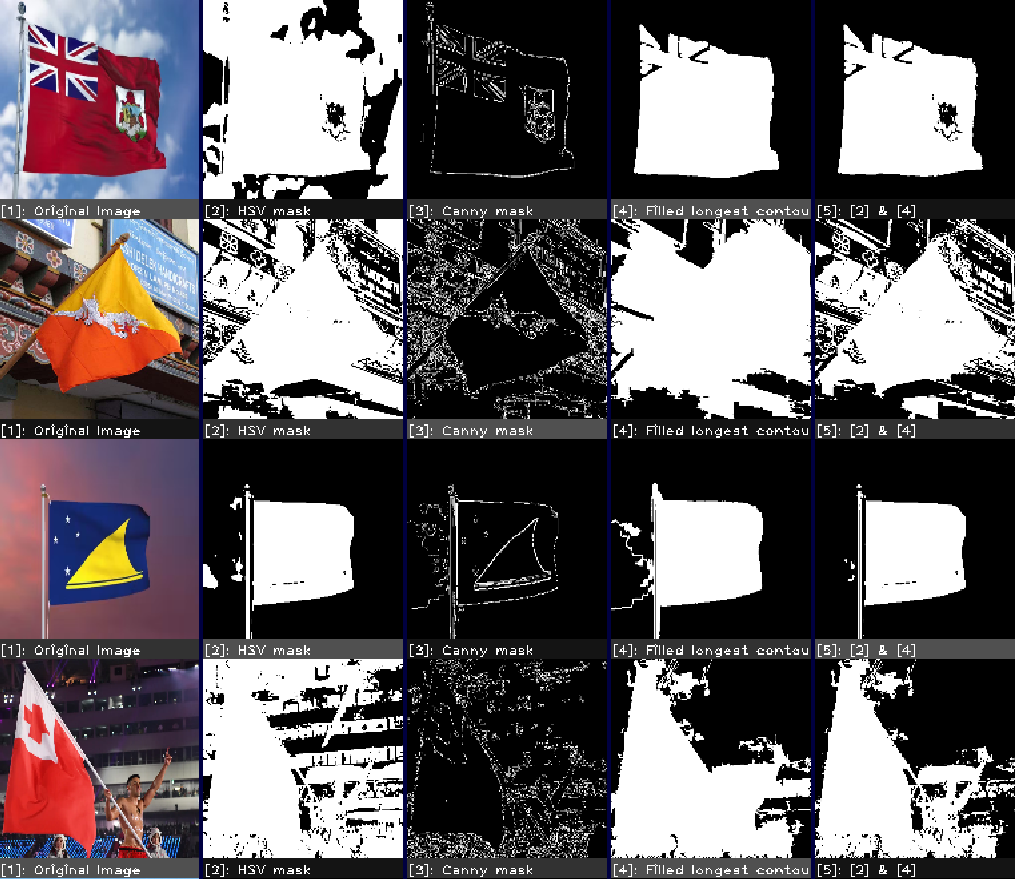


Figure 3.1.4. Binary and operation between the colour mask and the filled longest edge mask.

After getting the colour mask with some of the background noises removed, there is still an additional step to remove more background noises. The colour mask with some background noises removed is deducted with the mask containing the edges segmented in the previous step. This step will cut the noisy background into very small pieces. This step will also cut the flag, but the flags will not be cut into small pieces like the background. Therefore, the background can be removed easily by finding the contours in the mask and removing the small ones. The threshold is set to 0.005, which means that any contour with an area smaller than 0.5% of the size of the image will be removed. After removing the small contours, the large contours which are more likely to be the flag are connected back using a very heavy dilation operation. The kernel used is a 3x3 kernel, and the number of iterations is set to 3. It is because the canny edge might also cut the pattern inside the flag to be very small pieces, so the large pieces require a heavy dilation to connect them back. Figure 3.1.5 shows some of the results after this operation.

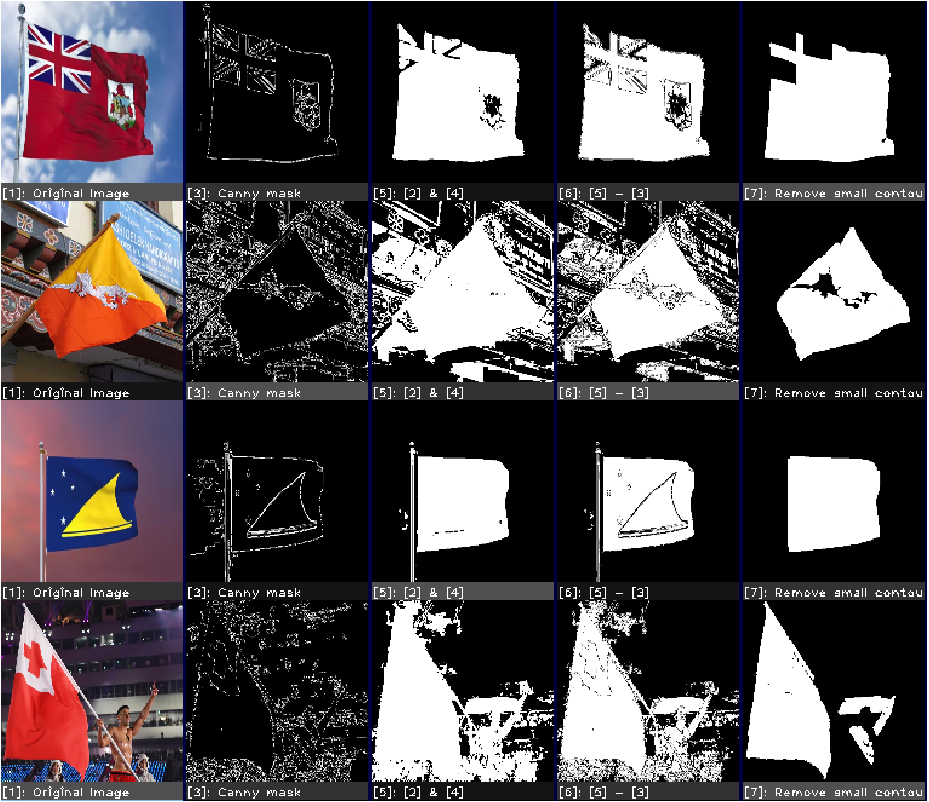


Figure 3.1.5. Removing very small noises in the image.

There could still be some large noises in the background after cutting, removing the small contours, and connecting back the flag. The large noises are removed by first finding the contours in the mask and leaving only the largest contour. The largest contour is most possibly the flag. Figure 3.1.6 shows the result after removing the large noises. A rectangular bounding box is then drawn to contain the largest contour in the mask. Figure 3.1.7 shows the whole segmentation flow.



Figure 3.1.6. Removing large noises in the image.

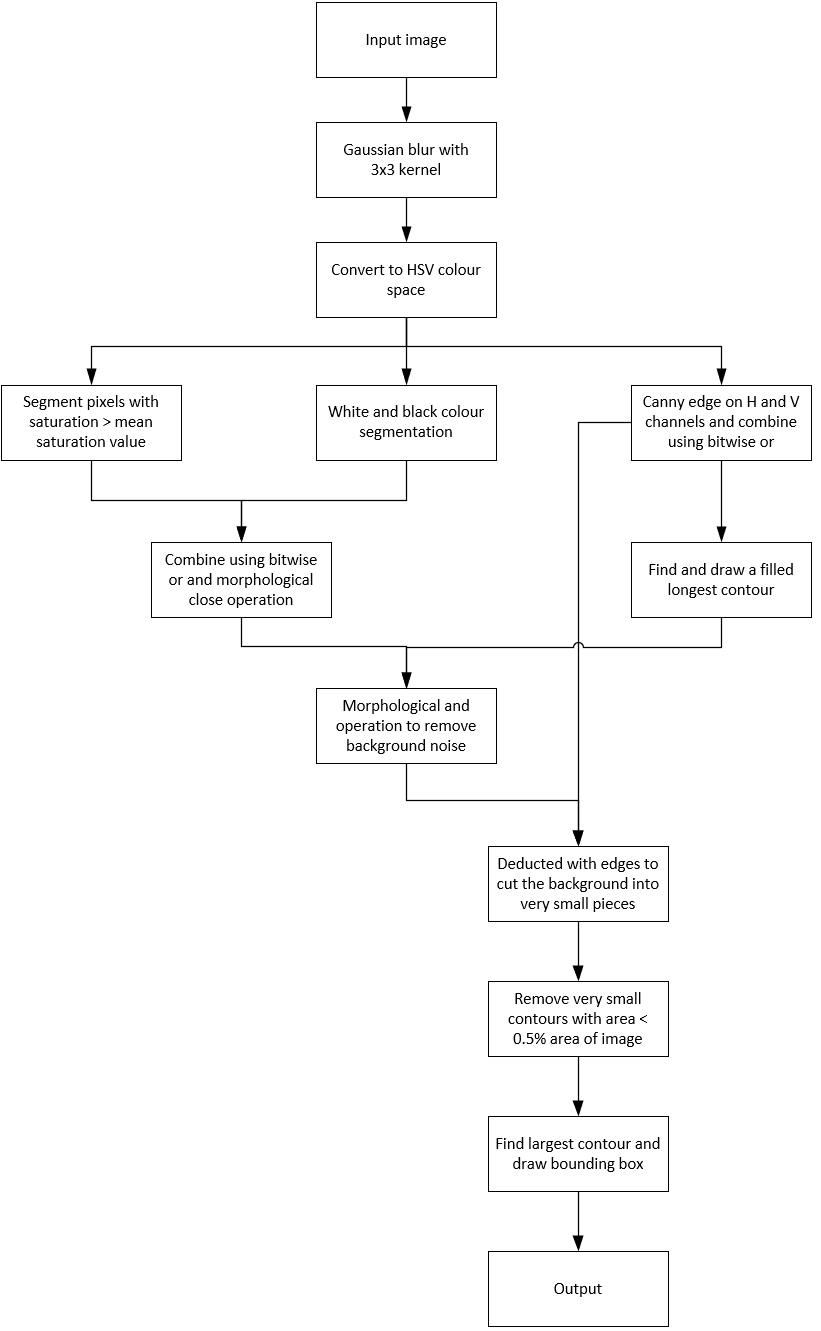


Figure 3.1.7. Segmentation flow 2.

**3.2 Classification**

We created the training dataset for the selected two rows of images. The training dataset for the selected two rows contains clear images from Set B and cropped real-world images. There is one clear image, and 10 cropped noisy images for each class for the selected rows. The training dataset for another two rows is adopted from another group. There is one clear image, and 7 cropped noisy images in each class for the adopted dataset. The training process is divided into four major steps – data pre-processing, feature extraction, normalisation, and fine-tuning.

**Data pre-processing**

In data pre-processing, the training images are first shuffled to add randomness to the dataset. Then, the training images are resized to 256x256 pixels and blurred using Gaussian blur with a 3x3 kernel to reduce noises in the images. The feature extraction extracts colour features of the images in HSV colour space. Thus, the images are also converted to HSV colour space in this step.

**Feature extraction**

Pure colour-based feature extraction method is used to get the features out of the image. Before extracting the features, the image is first divided into nine vertical regions and five horizontal regions. The features extracted in the vertical regions are independent of each other. On the other hand, the features generated in the horizontal regions can combine with one another to handle the situation where the flag is horizontally flipped. The features generated in region 1 are combined with region 5, region 2 is combined with region 4, and region 3 does not combine with others. The features generated by this method would remain the same no matter if the flag is horizontally flipped or not. Figure 3.2.1 shows the theory, and figure 3.2.2 shows an example of horizontally flipped country flag. Each colour represents one set of features generated.



Figure 3.2.1. Vertical and horizontal separations of the images.

Figure 3.2.2. Horizontally flipped country flag.

Besides, there are two additional regions to segment the features. The first region is at the middle 0.3 of image width \* image height, and the second is at the middle 0.3 of image height \* image width. Figure 3.2.3 illustrates the two additional regions, where the red colour is the interested region. Indeed, the intention of separating the images into many regions is to extract the patterns on the flag by the colour of the patterns. Smaller separations could extract smaller patterns while larger separations could extract larger patterns, and that is the reason for having the additional two bigger middle regions to generate the features. The middle regions are selected because they contain less noise than the sides in a noisy image as shown in figure 3.2.2.

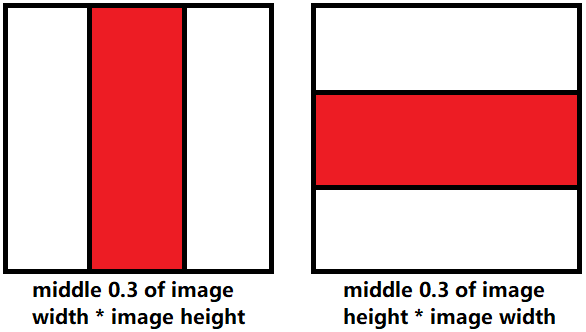


Figure 3.2.3. Two additional middle areas.

The colour features are extracted by setting bins, counting the number of pixels falling inside each bin, and further dividing the number by the region's area. There are two sets of bins to separate the coloured pixels and the black, white pixels. The coloured pixels are defined as those with both saturation and value larger than 50 in the HSV colour space. For coloured pixels set, there are ten bins in total. The range for value and saturation is the same for all bins, which are from 50 to 255. The only difference between the bins is the range of hue. Since there are ten bins in total, the size of hue for each bin is 18. Therefore, the range for bin 1 is from 0 to 17, bin 2 is 18 to 35, until the last bin is 162 to 180. On the other hand, there are five bins in total for black and white pixels set. The range of hue and saturation for each bin is the same, which are 0 to 180 for hue and 0 to 49 for saturation. The only difference between the bins is the range of value. Since there are five bins in total, the range for bin 1 is 0 to 50, bin 2 is 51 to 101, until the last bin is 204 to 255.

**Normalisation**

The mean and standard deviation of the training features are calculated, and the training features are then subtracted by the mean and divided by the standard deviation calculated. The mean and standard deviation obtained are saved into a text file because the data in the testing set must also be normalised using the mean and standard deviation of the training data.

**Fine tuning**

The fine-tuning process is primarily to fine-tune the feature extraction process instead of fine-tuning the SVM model. It is because generating useful features can boost the accuracy to a higher level than merely adjusting the SVM model's parameters. The parameters used in the feature extraction process that can be tweaked include:

* the number of bins for coloured pixels.
* the number of bins for black and white pixels.
* the number of separations in vertical and horizontal areas.
* the vertical and horizontal middle areas.

The goal is to get the accuracy as high as possible. The final model is trained using the combination of parameters that could give the best accuracy.

**Chapter 4: Experiment and Simulation**

**4.1 Implementation issues and challenges during segmentation**

The first issue is that the noise reduction strategy where the mask containing the flag is deducted by the mask generated by the canny edge might cut the flag into two halves if the pattern on the flag is large and extremely complicated. Figure 4.1.1 shows some examples. However, the strategy still brings more benefits than harm because very few images have large, complicated patterns like the flags in figure 4.1.1. Therefore, the strategy is still adopted in the segmentation process.

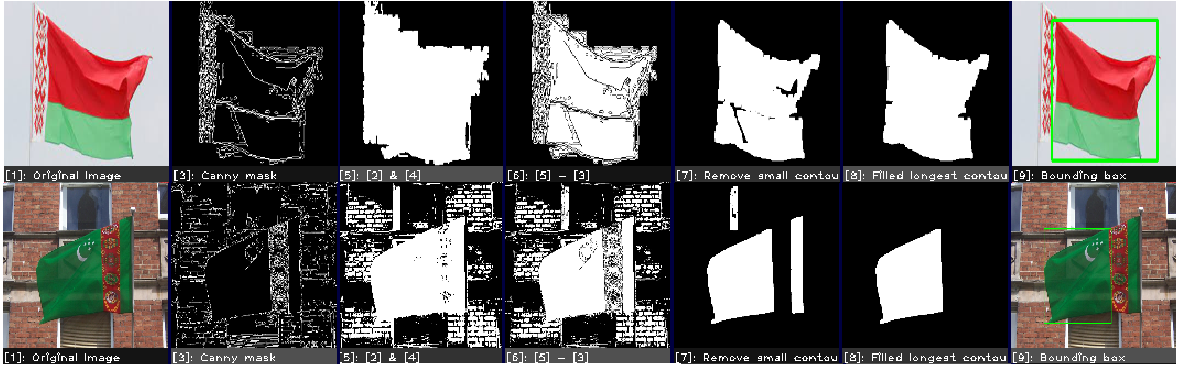


Figure 4.1.1. Noise reduction strategy cut the flag because of complicated pattern.

The second issue is also caused by the noise reduction strategy. When the flag is wrinkled, the canny edge will see the wrinkles as edges. When the mask containing the flag is deducted by the edges generated, the flag will lose many details at the wrinkled part. If the flag is too wrinkled, some parts of the flag will be assumed as noises and get removed. It will further cause a less accurate segmentation result. Figure 4.1.2 shows some examples.

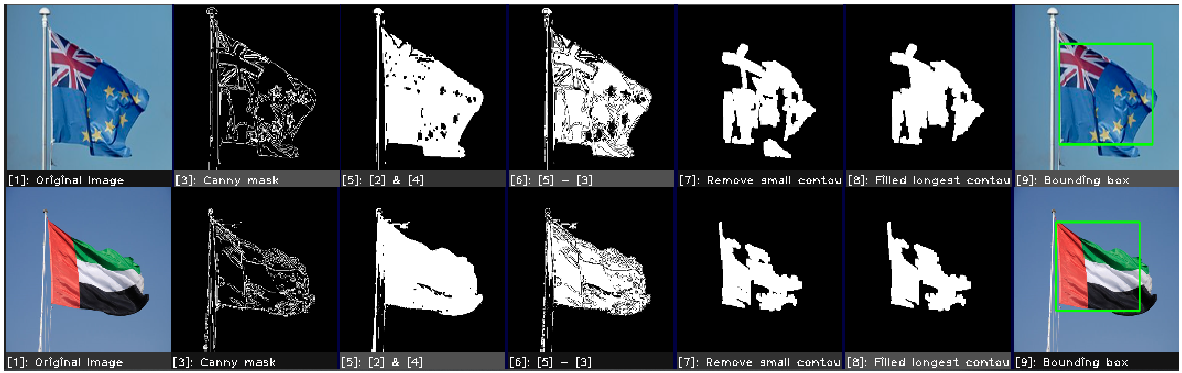


Figure 4.1.2. Noise reduction strategy cut the flag because of wrinkled flag.

**4.2 Implementation issues and challenges during classification (Jong Qian Biao)**

The recognition model is very weak at differentiating very similar flags. Figure 4.2.1 shows some of the extremely similar flags. Although these flags have some patterns that differ from each other, but the noises in noisy images could easily cheat the recognition model to predict them as a wrong category because the recognition model might see the patterns as noises as well. Furthermore, the recognition model is trained using cropped noisy images from the real world, and the dataset is relatively small. The recognition model might need more training images to differentiate the patterns on the flag with noises.

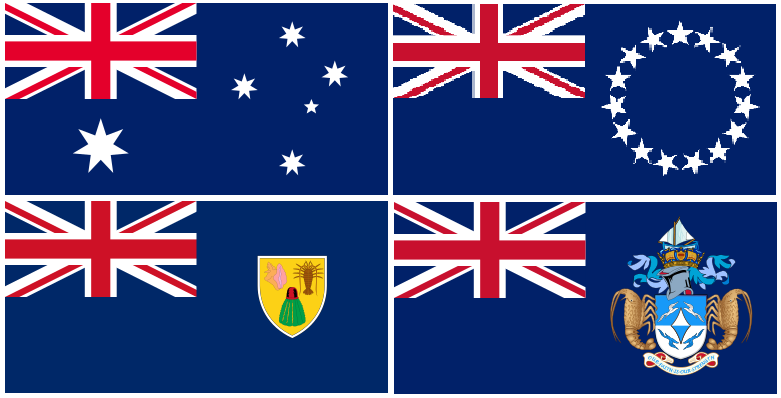


Figure 4.2.1. Extremely similar flags.

**Chapter 5: System evaluation and discussion**

**5.1 System performance definition**

**Segmentation**

We use intersection over union (IOU) to evaluate our proposed segmentation method. IOU is defined as the percentage of the overlapping area over the segmented region and the ground truth. The equation of intersection over union is R∩G/(R+G–R∩G), where R represents the segmented region and G represents the ground truth. The concept of IOU is illustrated in figure 5.1.1. The average segmentation accuracy on the whole dataset is obtained by adding up all the accuracy of a single image and dividing by the total number of images.

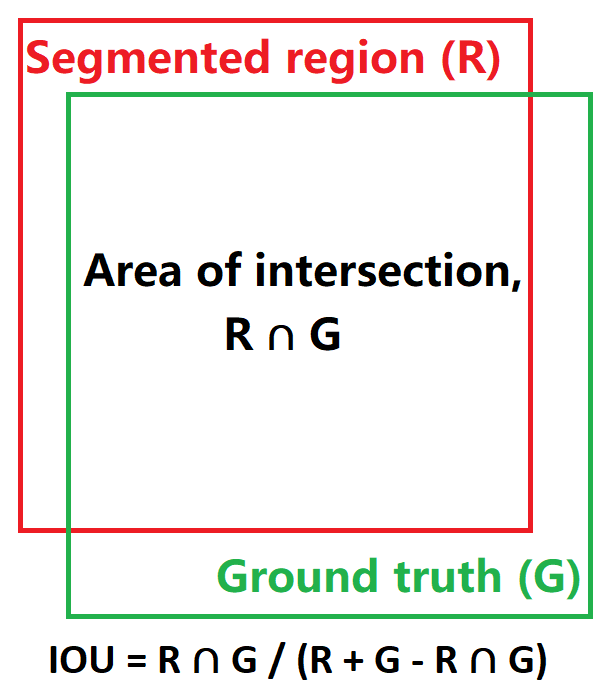


Figure 5.1 1 Intersection over union.

**Recognition**

The unit used to measure the recognition performance is accuracy, which is defined as the number of correct predictions divided by the total number of samples.

**5.2 Results**

The segmentation and classification accuracy are as listed in the table below. The segmentation flows can get 100% segmentation accuracy for the cropped images in Set A, B and D, and 90.04% segmentation accuracy for real-world images in set C. For classification, the accuracy is 87.50% for set A, 93.75% for set B, 90.63% for set C, and 96.88% for set D.

Table 5.2.1. Segmentation accuracy and classification accuracy.

|  |  |  |
| --- | --- | --- |
| Set | Segmentation accuracy | Classification accuracy |
| A | 100% | 87.50% |
| B | 100% | 93.75% |
| C | 90.04% | 90.63% |
| D | 100% | 96.88% |

**Chapter 6: Conclusion and recommendation**

**6.1 Conclusion**

To conclude, we proposed a complete flow for fine segmentation of flags that tackles different problems, such as illumination and traffic sign conditions. Our proposed method works well on the cropped flag dataset, but more work must be done to improve our methods to adapt better in real environments. Although we can get a high segmentation and classification accuracy, the strategies would not work well under certain conditions in real life.

**6.2 Novelty of work**

This project has proposed a very effective strategy to remove noises from the background, which is cutting the binary mask using edges generated by canny edge operation and removing the small background noises by removing very small contours. The large noises can then be removed by comparing the size of the remaining contours. The largest contour should be the flag, and others can be removed. This strategy is very useful, especially when segmenting a flag with a very complicated background.

This project also proposed a novel colour feature extraction strategy that allows the SVM model to remember the colours on the flag in separated regions. The features are generated by setting bins and counting the number of pixels falling inside each bin and dividing the number of pixels by the area of the region. The features extracted are unaffected by whether a flag is flipped horizontally, as it is very common to happen on the flags when blowing by the wind.

**6.3 Future work**

The noise removing strategy can be further improved so that it does not remove the patterns and wrinkles inside the flag, or some additional strategies can be explored to recover the removed part. The feature extraction part could also be improved so that the SVM model can differentiate between very similar flags.

**Appendix**

1. **Segmentation.cpp**

#include <opencv2/opencv.hpp>

#include <opencv2/highgui/highgui.hpp>

#include <opencv2/core.hpp>

#include <opencv2/imgproc.hpp>

#include <opencv2/highgui.hpp>

#include <opencv2/ml.hpp>

#include <iostream>

#include <fstream>

#include <vector>

#include <algorithm>

using namespace std;

using namespace cv;

using namespace cv::ml;

void segmentate1(vector<Mat> images, vector<String> fileNames, string setName, vector<String> classes, Ptr<SVM> svm, Scalar mean, Scalar stdDev);

void segmentate2(vector<Mat> images, vector<String> fileNames, string setName, vector<String> classes, Ptr<SVM> svm, Scalar mean, Scalar stdDev, vector<int\*> coordinates);

void avgSaturation(Mat source, Mat& result);

void cannySeg(Mat source, Mat& result);

void findLargestContour(Mat source, Mat& noFillResult, Mat& filledResult);

void removeSmallContours(Mat source, Mat& result, double threshold);

void boundingBox(Mat source, Mat& result, Rect& rect);

void colourSegHSV(Mat source, Mat& result);

void FeatureExtraction(Mat image, Mat& features);

#define SEPH 3

#define SEPV 15

void createWindowPartition(Mat srcI, Mat& largeWin, Mat win[], Mat legends[], int noOfImagePerCol = 1,

int noOfImagePerRow = 1, int sepH = SEPH, int sepV = SEPV);

Mat kernel = (Mat\_<unsigned char>(3, 3)

<< 1, 1, 1,

1, 1, 1,

1, 1, 1);

int main()

{

Ptr<SVM> SVM = SVM::load("Trained models\\SVM - all rows.xml");

ifstream readNorm("Trained models\\Norm - all rows.txt");

double meantemp, stdDevtemp;

readNorm >> meantemp;

readNorm >> stdDevtemp;

readNorm.close();

Scalar mean(meantemp), stdDev(stdDevtemp);

ifstream readClasses("Trained models\\Labels - all rows.txt");

vector<String> classes = {};

String className;

while (getline(readClasses, className)) {

classes.push\_back(className);

}

auto start = chrono::high\_resolution\_clock::now();

// set A folder location: Inputs/SetA

system("dir \"Inputs\\SetA\\\*.png\" /b > dirList.txt");

ifstream setA("dirList.txt");

vector<Mat> imagesA = {};

vector<String> fileNamesA = {};

string fileNameA;

while (getline(setA, fileNameA)) {

imagesA.push\_back(imread("Inputs/SetA/" + fileNameA));

fileNamesA.push\_back(fileNameA);

}

setA.close();

segmentate1(imagesA, fileNamesA, "setA", classes, SVM, mean, stdDev);

// set B folder location: Inputs/SetB

system("dir \"Inputs\\SetB\\\*.png\" /b > dirList.txt");

ifstream setB("dirList.txt");

vector<Mat> imagesB = {};

vector<String> fileNamesB = {};

string fileNameB;

while (getline(setB, fileNameB)) {

imagesB.push\_back(imread("Inputs/SetB/" + fileNameB));

fileNamesB.push\_back(fileNameB);

}

setB.close();

segmentate1(imagesB, fileNamesB, "setB", classes, SVM, mean, stdDev);

// set C folder location: Inputs/SetC

system("dir \"Inputs\\SetC\\\*.png\" /b > dirList.txt");

ifstream setC("dirList.txt");

vector<Mat> imagesC = {};

vector<String> fileNamesC = {};

string fileNameC;

while (getline(setC, fileNameC)) {

imagesC.push\_back(imread("Inputs/SetC/" + fileNameC));

fileNamesC.push\_back(fileNameC);

}

setC.close();

ifstream readCoordinates("Inputs\\SetC\\Coordinations.txt");

vector<int\*> coordinates = {};

for (int i = 0; i < imagesC.size(); i++) {

int\* coordinate = new int[4];

readCoordinates >> coordinate[0];

readCoordinates >> coordinate[1];

readCoordinates >> coordinate[2];

readCoordinates >> coordinate[3];

coordinates.push\_back(coordinate);

}

segmentate2(imagesC, fileNamesC, "setC", classes, SVM, mean, stdDev, coordinates);

// set D folder location: Inputs/SetD

system("dir \"Inputs\\SetD\\\*.png\" /b > dirList.txt");

ifstream setD("dirList.txt");

vector<Mat> imagesD = {};

vector<String> fileNamesD = {};

string fileNameD;

while (getline(setD, fileNameD)) {

imagesD.push\_back(imread("Inputs/SetD/" + fileNameD));

fileNamesD.push\_back(fileNameD);

}

setD.close();

segmentate1(imagesD, fileNamesD, "setD", classes, SVM, mean, stdDev);

system("del dirList.txt");

auto stop = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::milliseconds>(stop - start).count();

cout << "Time taken " << duration << "ms" << endl;

int totalImage = imagesA.size() + imagesB.size() + imagesC.size() + imagesD.size();

cout << "Total image = " << totalImage << endl;

cout << "Time taken per image = " << duration / totalImage << "ms" << endl;

return 0;

}

void segmentate1(vector<Mat> images, vector<String> fileNames, string setName, vector<String> classes, Ptr<SVM> svm, Scalar mean, Scalar stdDev) {

Size outputSize = Size(150, 150);

Mat results;

const int noOfImagePerCol = 100, noOfImagePerRow = 3;

Mat win[noOfImagePerCol \* noOfImagePerRow];

Mat legends[noOfImagePerCol \* noOfImagePerRow];

int totalResultImageCount = 0;

int correctPrediction = 0;

cout << setName << " prediction result" << endl;

for (int i = 0; i < images.size(); i++) {

int currentIndex = i % 100;

if (currentIndex == 0) {

createWindowPartition(Mat(outputSize, CV\_8UC3), results, win, legends, noOfImagePerCol, noOfImagePerRow);

}

// show original image

resize(images[i], win[currentIndex \* noOfImagePerRow], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow], "[1]: Original Image", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// segment out more saturated regions

Mat hsvMask;

colourSegHSV(images[i], hsvMask);

resize(hsvMask, win[currentIndex \* noOfImagePerRow + 1], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 1], "[2]: HSV mask", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// draw bounding box

Mat finalMask;

Rect rect;

images[i].copyTo(finalMask);

boundingBox(hsvMask, finalMask, rect);

resize(finalMask, win[currentIndex \* noOfImagePerRow + 2], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 2], "[3]: Bounding box", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

if (currentIndex == 99 || i == images.size() - 1) {

imwrite(setName + " results " + to\_string(totalResultImageCount++) + ".png", results(Range(0, outputSize.height \* currentIndex + SEPV \* currentIndex), Range::all()));

}

Mat feature;

FeatureExtraction(images[i], feature);

subtract(feature, mean, feature);

divide(feature, stdDev, feature);

String actualLabel = fileNames[i].substr(0, fileNames[i].size() - 4);

int actualLabelIndex = 0;

for (int j = 0; j < classes.size(); j++) {

if (actualLabel == classes[j]) {

actualLabelIndex = j;

}

}

int predictedLabelIndex = svm->predict(feature);

String predictedLabel = classes[predictedLabelIndex];

if (actualLabelIndex == predictedLabelIndex) {

correctPrediction++;

}

cout << "Actual: " << actualLabelIndex << "\tPredicted: " << predictedLabelIndex << "\t(" << actualLabel << " as " << predictedLabel << ")" << endl;

}

cout << "Recognition Accuracy: " << correctPrediction / (double)images.size() << endl;

cout << endl << endl;

}

void segmentate2(vector<Mat> images, vector<String> fileNames, string setName, vector<String> classes, Ptr<SVM> svm, Scalar mean, Scalar stdDev, vector<int\*> coordinates) {

Size outputSize = Size(150, 150);

Mat results;

const int noOfImagePerCol = 100, noOfImagePerRow = 11;

Mat win[noOfImagePerCol \* noOfImagePerRow];

Mat legends[noOfImagePerCol \* noOfImagePerRow];

int totalResultImageCount = 0;

int correctPrediction = 0;

double total\_accuracy = 0;

cout << setName << " prediction result" << endl;

for (int i = 0; i < images.size(); i++) {

int currentIndex = i % 100;

if (currentIndex == 0) {

createWindowPartition(Mat(outputSize, CV\_8UC3), results, win, legends, noOfImagePerCol, noOfImagePerRow);

}

// show original image

resize(images[i], win[currentIndex \* noOfImagePerRow], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow], "[1]: Original Image", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// segment out more saturated regions

Mat saturationMask;

avgSaturation(images[i], saturationMask);

resize(saturationMask, win[currentIndex \* noOfImagePerRow + 1], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 1], "[2]: HSV mask", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// canny edge based on hue and saturation

Mat cannyMask, dilatedCannyMask;

cannySeg(images[i], cannyMask);

dilate(cannyMask, dilatedCannyMask, kernel);

resize(cannyMask, win[currentIndex \* noOfImagePerRow + 2], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 2], "[3]: Canny mask", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// find longest edge

Mat edgeLargestContourMaskNoFill, edgeLargestContourMaskFilled;

findLargestContour(dilatedCannyMask, edgeLargestContourMaskNoFill, edgeLargestContourMaskFilled);

resize(edgeLargestContourMaskFilled, win[currentIndex \* noOfImagePerRow + 3], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 3], "[4]: Filled longest contour in [3]", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// saturation mask & filled longest contour formed by canny edge

Mat satAndFilled = saturationMask & edgeLargestContourMaskFilled;

resize(satAndFilled, win[currentIndex \* noOfImagePerRow + 4], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 4], "[5]: [2] & [4]", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// cut the mask found with edge

Mat cutWithEdge = satAndFilled - cannyMask;

resize(cutWithEdge, win[currentIndex \* noOfImagePerRow + 5], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 5], "[6]: [5] - [3]", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// remove very small contours which are noises

Mat cleanSatAndFilled;

removeSmallContours(cutWithEdge, cleanSatAndFilled, .005);

resize(cleanSatAndFilled, win[currentIndex \* noOfImagePerRow + 6], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 6], "[7]: Remove small contours", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// find largest contour

Mat finalLargestContourMaskNoFill, finalLargestContourMaskFilled;

findLargestContour(cleanSatAndFilled, finalLargestContourMaskNoFill, finalLargestContourMaskFilled);

resize(finalLargestContourMaskFilled, win[currentIndex \* noOfImagePerRow + 7], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 7], "[8]: Filled longest contour in [7]", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// draw bounding box

Mat finalMask;

Rect rect;

images[i].copyTo(finalMask);

boundingBox(finalLargestContourMaskFilled, finalMask, rect);

resize(finalMask, win[currentIndex \* noOfImagePerRow + 8], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 8], "[9]: Bounding box", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// crop the image for recognition

Mat toRecog = images[i](rect);

resize(toRecog, win[currentIndex \* noOfImagePerRow + 9], outputSize, INTER\_LINEAR);

putText(legends[currentIndex \* noOfImagePerRow + 9], "[10]: For recognition", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

// show segmentation accuracy

int\* coordinate = coordinates[i];

double area1 = rect.area();

double area2 = (coordinate[2] - coordinate[0]) \* (coordinate[3] - coordinate[1]);

double x\_diff = max({ 0, min({coordinate[2], rect.x + rect.width}) }) - max({ coordinate[0], rect.x });

double y\_diff = max({ 0, min({coordinate[3], rect.y + rect.height}) }) - max({coordinate[1], rect.y});

double overlap = x\_diff \* y\_diff;

double area = area1 + area2 - overlap;

double accurracy = overlap / area;

total\_accuracy += accurracy;

putText(win[currentIndex \* noOfImagePerRow + 10], "Accuracy: " + to\_string(accurracy), Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

putText(legends[currentIndex \* noOfImagePerRow + 10], "[11]: Segmentation result", Point(1, 12), 1, .7, Scalar(255, 255, 255), 1);

//recognise the flag

Mat feature;

FeatureExtraction(toRecog, feature);

subtract(feature, mean, feature);

divide(feature, stdDev, feature);

String actualLabel = fileNames[i].substr(0, fileNames[i].size() - 4);

int actualLabelIndex = 0;

for (int j = 0; j < classes.size(); j++) {

if (actualLabel == classes[j]) {

actualLabelIndex = j;

}

}

int predictedLabelIndex = svm->predict(feature);

String predictedLabel = classes[predictedLabelIndex];

if (actualLabelIndex == predictedLabelIndex) {

correctPrediction++;

}

cout << "Actual: " << actualLabelIndex << "\tPredicted: " << predictedLabelIndex << "\t(" << actualLabel << " as " << predictedLabel << ")" << endl;

if (currentIndex == 99 || i == images.size() - 1) {

imwrite(setName + " results " + to\_string(totalResultImageCount++) + ".png", results(Range(0, outputSize.height \* (currentIndex + 1) + SEPV \* (currentIndex + 1)), Range::all()));

}

}

cout << "Recognition Accuracy: " << correctPrediction / (double)images.size() << endl;

cout << "Segmentation Accuracy: " << total\_accuracy / (double)images.size() << endl;

cout << endl << endl;

}

void avgSaturation(Mat source, Mat& result) {

// blur the image

Mat blur;

GaussianBlur(source, blur, Size(3, 3), 0);

// convert to hsv colour space

Mat hsv;

cvtColor(blur, hsv, COLOR\_BGR2HSV);

// split into H, S and V channels

vector<Mat> channels;

split(hsv, channels);

// find the mean saturation value

Scalar avg\_saturation = mean(channels[1]);

// Ssegment out colours with high (larger than mean) saturation only

Scalar saturationStart = Scalar(0, avg\_saturation[0], 0), saturationEnd = Scalar(180, 255, 255);

inRange(hsv, saturationStart, saturationEnd, result);

// Segment out black and white colours

Scalar blackHSVStart = Scalar(0, 0, 0), blackHSVEnd = Scalar(180, 130, 50);

Scalar whiteHSVStart = Scalar(0, 0, 130), whiteHSVEnd = Scalar(180, 55, 255);

// thresholding

Mat blackMask, whiteMask;

inRange(hsv, blackHSVStart, blackHSVEnd, blackMask);

inRange(hsv, whiteHSVStart, whiteHSVEnd, whiteMask);

// adding into the result

result = result | blackMask | whiteMask;

// Morph close to fill small lines in the flag

morphologyEx(result, result, MORPH\_CLOSE, kernel);

// store the result

cvtColor(result, result, COLOR\_GRAY2BGR);

}

void cannySeg(Mat source, Mat& result) {

// blur the image

Mat blur;

GaussianBlur(source, blur, Size(3, 3), 0);

// convert to hsv colour space

Mat hsv;

cvtColor(blur, hsv, COLOR\_BGR2HSV);

// split into H, S, and V channels

vector<Mat> channels;

split(hsv, channels);

// Apply Canny edge detection to the hue and saturation channel

Mat edges, result1, result2;

Canny(channels[0], result1, 1, 200);

Canny(channels[1], result2, 1, 200);

// combine both masks

result = result1 | result2;

// store the result

cvtColor(result, result, COLOR\_GRAY2BGR);

}

void findLargestContour(Mat source, Mat& noFillResult, Mat& filledResult) {

// convert to single channel

Mat gray;

cvtColor(source, gray, COLOR\_BGR2GRAY);

// find contours

vector<vector<Point>> contours;

vector<Vec4i> hierarchy;

findContours(gray, contours, hierarchy, RETR\_EXTERNAL, CHAIN\_APPROX\_SIMPLE);

// get largest contour

double largestArea = 0;

int largestContourIndex = -1;

for (int i = 0; i < contours.size(); i++) {

double area = contourArea(contours[i]);

if (area > largestArea) {

largestArea = area;

largestContourIndex = i;

}

}

// draw the result

noFillResult = Mat::zeros(gray.size(), CV\_8UC3);

filledResult = Mat::zeros(gray.size(), CV\_8UC3);

drawContours(noFillResult, contours, largestContourIndex, Scalar(255, 255, 255), 1);

drawContours(filledResult, contours, largestContourIndex, Scalar(255, 255, 255), FILLED);

}

void removeSmallContours(Mat source, Mat &result, double threshold) {

// convert to single channel

Mat gray;

cvtColor(source, gray, COLOR\_BGR2GRAY);

// erode to remove small lines connecting the noises

erode(gray, gray, kernel);

// find contours

vector<vector<Point>> contours;

vector<Vec4i> hierarchy;

findContours(gray, contours, hierarchy, RETR\_EXTERNAL, CHAIN\_APPROX\_SIMPLE);

// keep the contours with size larger than the threshold only

double threshSize = threshold \* gray.rows \* gray.cols;

result = Mat::zeros(gray.size(), CV\_8UC3);

for (int i = 0; i < contours.size(); i++) {

double area = contourArea(contours[i]);

if (area > threshSize) {

drawContours(result, contours, i, Scalar(255, 255, 255), FILLED);

}

}

// dilate to connect back the large contours

dilate(result, result, kernel, Point(-1, -1), 3);

}

void boundingBox(Mat source, Mat& result, Rect& rect) {

// convert to single channel

Mat gray;

cvtColor(source, gray, COLOR\_BGR2GRAY);

// find contours

vector<vector<Point>> contours;

vector<Vec4i> hierarchy;

findContours(gray, contours, hierarchy, RETR\_EXTERNAL, CHAIN\_APPROX\_SIMPLE);

// get the bounding box of largest contour if exist

if (contours.size() > 0) {

rect = boundingRect(contours[0]);

// draw on result

rectangle(result, rect, Scalar(0, 255, 0), 3);

}

}

void colourSegHSV(Mat source, Mat& result) {

Mat blur, hsv, red1mask, red2mask, yellowmask, greenmask, bluemask, whitemask, blackmask, mask, segmentBGR;

Mat kernel = (Mat\_<unsigned char>(3, 3)

<< 1, 1, 1,

1, 1, 1,

1, 1, 1);

// define colour ranges in HSV

Scalar redStart1 = Scalar(0, 110, 140), redEnd1 = Scalar(15, 255, 255);

Scalar redStart2 = Scalar(165, 110, 140), redEnd2 = Scalar(180, 255, 255);

Scalar yellowStart1 = Scalar(15, 85, 50), yellowEnd1 = Scalar(40, 255, 255);

Scalar greenStart1 = Scalar(50, 100, 100), greenEnd1 = Scalar(85, 255, 255);

Scalar blueStart1 = Scalar(75, 70, 50), blueEnd1 = Scalar(140, 255, 255);

Scalar whiteStart1 = Scalar(0, 0, 190), whiteEnd1 = Scalar(180, 80, 255);

Scalar blackStart1 = Scalar(0, 0, 0), blackEnd1 = Scalar(180, 255, 45);

// blur source image using gaussian blur

medianBlur(source, blur, 3);

// convert source image into HSV colour space

cvtColor(blur, hsv, COLOR\_BGR2HSV);

// thresholding using the HSV colour ranges

inRange(hsv, redStart1, redEnd1, red1mask);

inRange(hsv, redStart2, redEnd2, red2mask);

inRange(hsv, yellowStart1, yellowEnd1, yellowmask);

inRange(hsv, greenStart1, greenEnd1, greenmask);

inRange(hsv, blueStart1, blueEnd1, bluemask);

inRange(hsv, whiteStart1, whiteEnd1, whitemask);

inRange(hsv, blackStart1, blackEnd1, blackmask);

mask = red1mask | red2mask | yellowmask | greenmask | bluemask | whitemask | blackmask;

// use morphological close operation to fill small gaps between segmented regions

dilate(mask, mask, kernel, Point(-1, -1), 3);

morphologyEx(mask, mask, MORPH\_CLOSE, kernel);

// store result

cvtColor(mask, result, COLOR\_GRAY2BGR);

}

void FeatureExtraction(Mat image, Mat& features) {

// Preprocessing

Size fixedSize = Size(256, 256);

resize(image, image, fixedSize, INTER\_LINEAR);

GaussianBlur(image, image, Size(3, 3), 0);

Mat hsv;

cvtColor(image, hsv, COLOR\_BGR2HSV);

// colour feature Extraction

int numOfBins1 = 10;

int minSat = 50;

int minValue = 50;

int numOfBins2 = 5;

int numOfSepsHorizontal = 5;

int numOfSepsVertical = 9;

double middleAreaVertical = 0.3;

double middleAreaHorizontal = 0.3;

// Count the number of pixels in each bin in each separation according to their Hue

int height = image.rows;

int width = image.cols;

int binSize1 = 180 / numOfBins1;

int sepSizeHorizontal = height / numOfSepsHorizontal;

int sepSizeVertical = width / numOfSepsVertical;

int totalSepsVertical = numOfSepsVertical / 2 + numOfSepsVertical % 2;

features = Mat(1, (numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + totalSepsVertical + 2), CV\_32F);

for (int i = 0; i < numOfBins1; i++) {

int startThresh = i \* binSize1;

int endThresh = startThresh + binSize1 - 1;

if (endThresh + binSize1 > 180) {

endThresh = 180;

}

Mat mask;

Scalar rangeStart(startThresh, minSat, minValue), rangeEnd(endThresh, 255, 255);

for (int j = 0; j < numOfSepsHorizontal; j++) {

int rowRangeStart = j \* sepSizeHorizontal;

int rowRangeEnd = rowRangeStart + sepSizeHorizontal;

if (rowRangeEnd + sepSizeHorizontal > height) {

rowRangeEnd = height;

}

inRange(hsv.rowRange(rowRangeStart, rowRangeEnd), rangeStart, rangeEnd, mask);

features.at<float>((numOfBins1 + numOfBins2) \* j + i) = countNonZero(mask) / (float)(width \* (rowRangeEnd - rowRangeStart));

}

Mat mask1, mask2;

for (int j = 0; j < totalSepsVertical; j++) {

int colRangeStart1 = j \* sepSizeVertical;

int colRangeEnd1 = colRangeStart1 + sepSizeVertical;

int colRangeStart2 = (numOfSepsVertical - j - 1) \* sepSizeVertical;

int colRangeEnd2 = colRangeStart2 + sepSizeVertical;

inRange(hsv.colRange(colRangeStart1, colRangeEnd1), rangeStart, rangeEnd, mask1);

inRange(hsv.colRange(colRangeStart2, colRangeEnd2), rangeStart, rangeEnd, mask2);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + j) + i) = (countNonZero(mask1) + countNonZero(mask2)) / (float)(height \* (colRangeEnd1 - colRangeStart1) \* 2);

}

int colRangeStart = width / 2 - width \* middleAreaVertical / 2;

int colRangeEnd = width / 2 + width \* middleAreaVertical / 2;

inRange(hsv.colRange(colRangeStart, colRangeEnd), rangeStart, rangeEnd, mask);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + totalSepsVertical) + i) = countNonZero(mask) / (float)(height \* (colRangeEnd - colRangeStart));

int rowRangeStart = height / 2 - height \* middleAreaHorizontal / 2;

int rowRangeEnd = height / 2 + height \* middleAreaHorizontal / 2;

inRange(hsv.rowRange(rowRangeStart, rowRangeEnd), rangeStart, rangeEnd, mask);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + totalSepsVertical + 1) + i) = countNonZero(mask) / (float)(width \* (rowRangeEnd - rowRangeStart));

}

// for black and white according to their value

int binSize2 = 255 / numOfBins2;

for (int i = 0; i < numOfBins2; i++) {

int startThresh = i \* binSize2;

int endThresh = startThresh + binSize2 - 1;

if (endThresh + binSize2 > 255) {

endThresh = 255;

}

Mat mask;

Scalar rangeStart(0, 0, startThresh), rangeEnd(180, minSat - 1, endThresh);

for (int j = 0; j < numOfSepsHorizontal; j++) {

int rowRangeStart = j \* sepSizeHorizontal;

int rowRangeEnd = rowRangeStart + sepSizeHorizontal;

if (rowRangeEnd + sepSizeHorizontal > height) {

rowRangeEnd = height;

}

inRange(hsv.rowRange(rowRangeStart, rowRangeEnd), startThresh, endThresh, mask);

features.at<float>((numOfBins1 + numOfBins2) \* j + numOfBins1 + i) = countNonZero(mask) / (float)(width \* (rowRangeEnd - rowRangeStart));

}

Mat mask1, mask2;

for (int j = 0; j < totalSepsVertical; j++) {

int colRangeStart1 = j \* sepSizeVertical;

int colRangeEnd1 = colRangeStart1 + sepSizeVertical;

int colRangeStart2 = (numOfSepsVertical - j - 1) \* sepSizeVertical;

int colRangeEnd2 = colRangeStart2 + sepSizeVertical;

inRange(hsv.colRange(colRangeStart1, colRangeEnd1), rangeStart, rangeEnd, mask1);

inRange(hsv.colRange(colRangeStart2, colRangeEnd2), rangeStart, rangeEnd, mask2);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + j) + numOfBins1 + i) = (countNonZero(mask1) + countNonZero(mask2)) / (float)(height \* (colRangeEnd1 - colRangeStart1) \* 2);

}

int colRangeStart = width / 2 - width \* middleAreaVertical / 2;

int colRangeEnd = width / 2 + width \* middleAreaVertical / 2;

inRange(hsv.colRange(colRangeStart, colRangeEnd), rangeStart, rangeEnd, mask);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + totalSepsVertical) + numOfBins1 + i) = countNonZero(mask) / (float)(height \* (colRangeEnd - colRangeStart));

int rowRangeStart = height / 2 - height \* middleAreaHorizontal / 2;

int rowRangeEnd = height / 2 + height \* middleAreaHorizontal / 2;

inRange(hsv.rowRange(rowRangeStart, rowRangeEnd), rangeStart, rangeEnd, mask);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + totalSepsVertical + 1) + numOfBins1 + i) = countNonZero(mask) / (float)(width \* (rowRangeEnd - rowRangeStart));

}

}

void createWindowPartition(Mat srcI, Mat& largeWin, Mat win[], Mat legends[], int noOfImagePerCol,

int noOfImagePerRow, int sepH, int sepV) {

// 1st input: source input image

// 2nd: the created larger window

// 3th: means to access each sub window

// 4th: means to access each legend window

// 5rd, 6th: Obvious

// 7th: separating space between 2 images in horizontal direction

// 8th: separating space between 2 images in vertial direction

int rows = srcI.rows, cols = srcI.cols, winI = 0, winsrcI = 0;

Size sRXC((cols + sepH) \* noOfImagePerRow - sepH, (rows + sepV) \* noOfImagePerCol),

s(cols, sepV);

largeWin = Mat::ones(sRXC, srcI.type()) \* 64;

for (int i = 0; i < noOfImagePerCol; i++)

for (int j = 0; j < noOfImagePerRow; j++)

win[winI++] = largeWin(Range((rows + sepV) \* i, (rows + sepV) \* i + rows),

Range((cols + sepH) \* j, (cols + sepH) \* j + cols));

for (int bg = 20, i = 0; i < noOfImagePerCol; i++)

for (int j = 0; j < noOfImagePerRow; j++) {

legends[winsrcI] = largeWin(Range((rows + sepV) \* i + rows, (rows + sepV) \* (i + 1)),

Range((cols + sepH) \* j, (cols + sepH) \* j + cols));

legends[winsrcI] = Scalar(bg, bg, bg);

bg += 30;

if (bg > 80) bg = 20;

winsrcI++;

}

}

1. **Recognition.cpp**

#include <iostream>

#include <fstream>

#include <vector>

#include <random>

#include <algorithm>

#include <opencv2/opencv.hpp>

#include <opencv2/core.hpp>

#include <opencv2/imgproc.hpp>

#include <opencv2/highgui.hpp>

#include <opencv2/ml.hpp>

using namespace std;

using namespace cv;

using namespace cv::ml;

void featureExtraction(Mat image, Mat& features);

int main1() {

string modelName = "all rows";

// train:test:val ratio

float valRatio = 0;

float testRatio = 0;

int minTreshold = 100; // minimum number of images required to give validation and test set at least one image

// read images and labels

vector<String> classes = {};

vector<int> classesImagesCount = {};

vector<Mat> images = {};

vector<int> labels = {};

{

cout << "Loading image..." << endl;

system("dir \"Inputs\\cropped recognition dataset\\\" /b > dirList.txt");

ifstream readClasses("dirList.txt");

string className;

while (getline(readClasses, className)) {

classes.push\_back(className);

string command = "dir \"Inputs\\cropped recognition dataset\\" + className + "\\\*.png\" /b > dirList2.txt";

system(command.c\_str());

ifstream readImages("dirList2.txt");

string imageName;

int label = classes.size() - 1;

int count = 0;

while (getline(readImages, imageName)) {

Mat image = imread("Inputs\\cropped recognition dataset\\" + className + "\\" + imageName);

images.push\_back(image);

labels.push\_back(label);

count++;

}

classesImagesCount.push\_back(count);

readImages.close();

}

readClasses.close();

system("del dirList.txt");

system("del dirList2.txt");

}

// split train, validation, and test sets according to the ratio

vector<Mat> trainImages = {};

vector<int> trainImagesLabels = {};

vector<Mat> valImages = {};

vector<int> valImagesLabels = {};

vector<Mat> testImages = {};

vector<int> testImagesLabels = {};

{

cout << "Splitting train/ val/ test sets..." << endl;

srand(42);

int allImagesStartIndex = 0;

for (int i = 0; i < classes.size(); i++) {

bool\* assigned = new bool[classesImagesCount[i]]{ false };

int currentClassValImages = classesImagesCount[i] \* valRatio;

if (currentClassValImages == 0 && classesImagesCount[i] > minTreshold) {

currentClassValImages++;

}

for (int j = 0; j < currentClassValImages; j++) {

int randint = rand() % classesImagesCount[i];

while (assigned[randint]) {

randint++;

if (randint >= classesImagesCount[i]) {

randint = 0;

}

}

assigned[randint] = true;

valImages.push\_back(images[allImagesStartIndex + randint]);

valImagesLabels.push\_back(labels[allImagesStartIndex + randint]);

}

int currentClassTestImages = classesImagesCount[i] \* testRatio;

if (currentClassTestImages == 0 && classesImagesCount[i] > minTreshold) {

currentClassTestImages++;

}

for (int j = 0; j < currentClassTestImages; j++) {

int randint = rand() % classesImagesCount[i];

while (assigned[randint]) {

randint++;

if (randint >= classesImagesCount[i]) {

randint = 0;

}

}

assigned[randint] = true;

testImages.push\_back(images[allImagesStartIndex + randint]);

testImagesLabels.push\_back(labels[allImagesStartIndex + randint]);

}

for (int j = 0; j < classesImagesCount[i]; j++) {

if (!assigned[j]) {

trainImages.push\_back(images[allImagesStartIndex + j]);

trainImagesLabels.push\_back(labels[allImagesStartIndex + j]);

}

}

allImagesStartIndex += classesImagesCount[i];

}

vector<int> shuffled = {};

for (int i = 0; i < trainImages.size(); i++) {

shuffled.push\_back(i);

}

mt19937 rng(42);

shuffle(shuffled.begin(), shuffled.end(), rng);

vector<Mat> shuffledImages = {};

vector<int> shuffledLabels = {};

for (int i = 0; i < shuffled.size(); i++) {

shuffledImages.push\_back(trainImages[shuffled[i]]);

shuffledLabels.push\_back(trainImagesLabels[shuffled[i]]);

}

trainImages = shuffledImages;

trainImagesLabels = shuffledLabels;

}

// feature extraction

Mat trainFeatures;

Mat trainLabels;

Mat valFeatures;

Mat valLabels;

Mat testFeatures;

Mat testLabels;

{

cout << "Extracting features..." << endl;

for (int i = 0; i < trainImages.size(); i++) {

Mat feature;

featureExtraction(trainImages[i], feature);

trainFeatures.push\_back(feature);

trainLabels.push\_back(trainImagesLabels[i]);

}

for (int i = 0; i < valImages.size(); i++) {

Mat feature;

featureExtraction(valImages[i], feature);

valFeatures.push\_back(feature);

valLabels.push\_back(valImagesLabels[i]);

}

for (int i = 0; i < testImages.size(); i++) {

Mat feature;

featureExtraction(testImages[i], feature);

testFeatures.push\_back(feature);

testLabels.push\_back(testImagesLabels[i]);

}

}

// normalisation

Scalar mean, stdDev;

meanStdDev(trainFeatures, mean, stdDev);

subtract(trainFeatures, mean, trainFeatures);

divide(trainFeatures, stdDev, trainFeatures);

if (valLabels.total() != 0) {

subtract(valFeatures, mean, valFeatures);

divide(valFeatures, stdDev, valFeatures);

}

if (testLabels.total() != 0) {

subtract(testFeatures, mean, testFeatures);

divide(testFeatures, stdDev, testFeatures);

}

// train and test the SVM classifier

cout << "Training classifier..." << endl;

Ptr<SVM> svm = SVM::create();

svm->setType(SVM::C\_SVC);

svm->setKernel(SVM::LINEAR);

svm->setTermCriteria(TermCriteria(TermCriteria::MAX\_ITER, 100, 1e-6));

svm->train(trainFeatures, ROW\_SAMPLE, trainLabels);

// save model and normalisation details

svm->save("Trained models\\SVM - " + modelName + ".xml");

ofstream file("Trained models\\Norm - " + modelName + ".txt");

file << mean[0] << endl;

file << stdDev[0] << endl;

file.close();

ofstream labelsFile("Trained models\\Labels - " + modelName + ".txt");

for (int i = 0; i < classes.size(); i++) {

labelsFile << classes[i] << endl;

}

labelsFile.close();

// Predict using the SVM classifier

cout << "Training set:" << endl;

int trainCorrectCount = 0;

Mat predictedTrainLabels;

svm->predict(trainFeatures, predictedTrainLabels);

for (int i = 0; i < trainLabels.total(); i++) {

cout << "Actual: " << trainLabels.at<int>(i) << "\tPredicted: " << predictedTrainLabels.at<float>(i) << "\t(" << classes[trainLabels.at<int>(i)] << " as " << classes[predictedTrainLabels.at<float>(i)] << ")" << endl;

if (predictedTrainLabels.at<float>(i) == trainLabels.at<int>(i)) {

trainCorrectCount++;

}

}

cout << "training accuracy: " << trainCorrectCount / (double)trainLabels.total() << endl;

cout << endl << endl;

if (valLabels.total() != 0) {

cout << "Validation set:" << endl;

int valCorrectCount = 0;

Mat predictedValLabels;

svm->predict(valFeatures, predictedValLabels);

for (int i = 0; i < valLabels.total(); i++) {

cout << "Actual: " << valLabels.at<int>(i) << "\tPredicted: " << predictedValLabels.at<float>(i) << "\t(" << classes[valLabels.at<int>(i)] << " as " << classes[predictedValLabels.at<float>(i)] << ")" << endl;

if (predictedValLabels.at<float>(i) == valLabels.at<int>(i)) {

valCorrectCount++;

}

}

cout << "validation accuracy: " << valCorrectCount / (double)valLabels.total() << endl;

cout << endl << endl;

}

if (testLabels.total() != 0) {

cout << "Test Set:" << endl;

int testCorrectCount = 0;

Mat predictedTestLabels;

svm->predict(testFeatures, predictedTestLabels);

for (int i = 0; i < testLabels.total(); i++) {

cout << "Actual: " << testLabels.at<int>(i) << "\tPredicted: " << predictedTestLabels.at<float>(i) << "\t(" << classes[testLabels.at<int>(i)] << " as " << classes[predictedTestLabels.at<float>(i)] << ")" << endl;

if (predictedTestLabels.at<float>(i) == testLabels.at<int>(i)) {

testCorrectCount++;

}

}

cout << "testing accuracy: " << testCorrectCount / (double)testLabels.total() << endl;

cout << endl << endl;

}

return 0;

}

void featureExtraction(Mat image, Mat &features) {

// Preprocessing

Size fixedSize = Size(256, 256);

resize(image, image, fixedSize, INTER\_LINEAR);

GaussianBlur(image, image, Size(3,3), 0);

Mat hsv;

cvtColor(image, hsv, COLOR\_BGR2HSV);

// colour feature Extraction

int numOfBins1 = 10;

int minSat = 50;

int minValue = 50;

int numOfBins2 = 5;

int numOfSepsHorizontal = 5;

int numOfSepsVertical = 9;

double middleAreaVertical = 0.3;

double middleAreaHorizontal = 0.3;

// Count the number of pixels in each bin in each separation according to their Hue

int height = image.rows;

int width = image.cols;

int binSize1 = 180 / numOfBins1;

int sepSizeHorizontal = height / numOfSepsHorizontal;

int sepSizeVertical = width / numOfSepsVertical;

int totalSepsVertical = numOfSepsVertical / 2 + numOfSepsVertical % 2;

features = Mat(1, (numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + totalSepsVertical + 2), CV\_32F);

for (int i = 0; i < numOfBins1; i++) {

int startThresh = i \* binSize1;

int endThresh = startThresh + binSize1 - 1;

if (endThresh + binSize1 > 180) {

endThresh = 180;

}

Mat mask;

Scalar rangeStart(startThresh, minSat, minValue), rangeEnd(endThresh, 255, 255);

for (int j = 0; j < numOfSepsHorizontal; j++) {

int rowRangeStart = j \* sepSizeHorizontal;

int rowRangeEnd = rowRangeStart + sepSizeHorizontal;

if (rowRangeEnd + sepSizeHorizontal > height) {

rowRangeEnd = height;

}

inRange(hsv.rowRange(rowRangeStart, rowRangeEnd), rangeStart, rangeEnd, mask);

features.at<float>((numOfBins1 + numOfBins2) \* j + i) = countNonZero(mask) / (float)(width \* (rowRangeEnd - rowRangeStart));

}

Mat mask1, mask2;

for (int j = 0; j < totalSepsVertical; j++) {

int colRangeStart1 = j \* sepSizeVertical;

int colRangeEnd1 = colRangeStart1 + sepSizeVertical;

int colRangeStart2 = (numOfSepsVertical - j - 1) \* sepSizeVertical;

int colRangeEnd2 = colRangeStart2 + sepSizeVertical;

inRange(hsv.colRange(colRangeStart1, colRangeEnd1), rangeStart, rangeEnd, mask1);

inRange(hsv.colRange(colRangeStart2, colRangeEnd2), rangeStart, rangeEnd, mask2);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + j) + i) = (countNonZero(mask1) + countNonZero(mask2)) / (float)(height \* (colRangeEnd1 - colRangeStart1) \* 2);

}

int colRangeStart = width / 2 - width \* middleAreaVertical / 2;

int colRangeEnd = width / 2 + width \* middleAreaVertical / 2;

inRange(hsv.colRange(colRangeStart, colRangeEnd), rangeStart, rangeEnd, mask);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + totalSepsVertical) + i) = countNonZero(mask) / (float)(height \* (colRangeEnd - colRangeStart));

int rowRangeStart = height / 2 - height \* middleAreaHorizontal / 2;

int rowRangeEnd = height / 2 + height \* middleAreaHorizontal / 2;

inRange(hsv.rowRange(rowRangeStart, rowRangeEnd), rangeStart, rangeEnd, mask);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + totalSepsVertical + 1) + i) = countNonZero(mask) / (float)(width \* (rowRangeEnd - rowRangeStart));

}

// for black and white according to their value

int binSize2 = 255 / numOfBins2;

for (int i = 0; i < numOfBins2; i++) {

int startThresh = i \* binSize2;

int endThresh = startThresh + binSize2 - 1;

if (endThresh + binSize2 > 255) {

endThresh = 255;

}

Mat mask;

Scalar rangeStart(0, 0, startThresh), rangeEnd(180, minSat - 1, endThresh);

for (int j = 0; j < numOfSepsHorizontal; j++) {

int rowRangeStart = j \* sepSizeHorizontal;

int rowRangeEnd = rowRangeStart + sepSizeHorizontal;

if (rowRangeEnd + sepSizeHorizontal > height) {

rowRangeEnd = height;

}

inRange(hsv.rowRange(rowRangeStart, rowRangeEnd), startThresh, endThresh, mask);

features.at<float>((numOfBins1 + numOfBins2) \* j + numOfBins1 + i) = countNonZero(mask) / (float)(width\*(rowRangeEnd - rowRangeStart));

}

Mat mask1, mask2;

for (int j = 0; j < totalSepsVertical; j++) {

int colRangeStart1 = j \* sepSizeVertical;

int colRangeEnd1 = colRangeStart1 + sepSizeVertical;

int colRangeStart2 = (numOfSepsVertical - j - 1) \* sepSizeVertical;

int colRangeEnd2 = colRangeStart2 + sepSizeVertical;

inRange(hsv.colRange(colRangeStart1, colRangeEnd1), rangeStart, rangeEnd, mask1);

inRange(hsv.colRange(colRangeStart2, colRangeEnd2), rangeStart, rangeEnd, mask2);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + j) + numOfBins1 + i) = (countNonZero(mask1) + countNonZero(mask2)) / (float)(height \* (colRangeEnd1 - colRangeStart1) \* 2);

}

int colRangeStart = width / 2 - width \* middleAreaVertical / 2;

int colRangeEnd = width / 2 + width \* middleAreaVertical / 2;

inRange(hsv.colRange(colRangeStart, colRangeEnd), rangeStart, rangeEnd, mask);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + totalSepsVertical) + numOfBins1 + i) = countNonZero(mask) / (float)(height \* (colRangeEnd - colRangeStart));

int rowRangeStart = height / 2 - height \* middleAreaHorizontal / 2;

int rowRangeEnd = height / 2 + height \* middleAreaHorizontal / 2;

inRange(hsv.rowRange(rowRangeStart, rowRangeEnd), rangeStart, rangeEnd, mask);

features.at<float>((numOfBins1 + numOfBins2) \* (numOfSepsHorizontal + totalSepsVertical + 1) + numOfBins1 + i) = countNonZero(mask) / (float)(width \* (rowRangeEnd - rowRangeStart));

}

}