```
#include "Utilities.h"
#include <iostream>
#include <fstream>
#include <list>
#include <experimental/filesystem> // C++-standard header file name
#include <filesystem> // Microsoft-specific implementation header file name
#include "opencv2/features2d.hpp"
#include <opencv2/ml.hpp>
#include "opencv2/objdetect.hpp"
#include "opencv2/imgcodecs.hpp"
using namespace std::experimental::filesystem::v1;
using namespace std;


// Sign must be at least 100x100
#define MINIMUM_SIGN_SIDE 100
#define MINIMUM_SIGN_AREA 10000
#define MINIMUM_SIGN_BOUNDARY_LENGTH 400
#define STANDARD_SIGN_WIDTH_AND_HEIGHT 200
// Best match must be 10% better than second best match
#define REQUIRED_RATIO_OF_BEST_TO_SECOND_BEST 1.1
// Located shape must overlap the ground truth by 80% to be considered a match
#define REQUIRED_OVERLAP 0.8
// Draw a passed line using a random colour if one is not provided

// Draw a passed line using a random colour if one is not provided
void DrawLine2(Mat result_image, Point point1, Point point2, Scalar passed_colour =
    -1.0)
{
        Scalar colour(rand() & 0xFF, rand() & 0xFF, rand() & 0xFF);
        line(result_image, point1, point2, (passed_colour.val[0] == -1.0) ? colour :
            passed_colour);
}

// Draw line segments delineated by end points
void DrawLines2(Mat result_image, vector<Vec4i> lines, Scalar passed_colour = -1.0)
{
        for (vector<cv::Vec4i>::const_iterator current_line = lines.begin();
                (current_line != lines.end()); current_line++)
        {
                Point point1((*current_line)[0], (*current_line)[1]);
                Point point2((*current_line)[2], (*current_line)[3]);
                DrawLine2(result_image, point1, point2, passed_colour);
        }
}
// Draw lines defined by rho and theta parameters
void DrawLines2(Mat result_image, vector<Vec2f> lines, Scalar passed_colour = -1.0)
{
        for (vector<cv::Vec2f>::const_iterator current_line = lines.begin();
                (current_line != lines.end()); current_line++)
        {
                float rho = (*current_line)[0];
                float theta = (*current_line)[1];
                // To avoid divide by zero errors we offset slightly from 0.0
                float cos_theta = (cos(theta) == 0.0) ? (float) 0.000000001 : (float
                    )cos(theta);
                float sin_theta = (sin(theta) == 0.0) ? (float) 0.000000001 : (float
                    )sin(theta);
```

```
            Point left((int)(rho / cos(theta)), 0);
            Point right((int)((rho - (result_image.rows - 1)*sin(theta)) / cos(
                theta)), (int)((result_image.rows - 1)));
            Point top(0, (int)(rho / sin(theta)));
            Point bottom((int)(result_image.cols - 1), (int)((rho - (
                result_image.cols - 1)*cos(theta)) / sin(theta)));
            Point* point1 = NULL;
            Point* point2 = NULL;
            if ((left.y >= 0.0) && (left.y <= (result_image.rows - 1)))
                    point1 = &left;
            if ((right.y >= 0.0) && (right.y <= (result_image.rows - 1)))
                    if (point1 == NULL)
                            point1 = &right;
                    else point2 = &right;
            if ((point2 == NULL) && (top.x >= 0.0) && (top.x <= (result_image.
                cols - 1)))
                    if (point1 == NULL)
                            point1 = &top;
                    else if ((point1->x != top.x) || (point1->y != top.y))
                            point2 = &top;
            if (point2 == NULL)
                    point2 = &bottom;
            DrawLine2(result_image, *point1, *point2, passed_colour);
    }
}



class ObjectAndLocation
{
public:
    ObjectAndLocation(string object_name, Point top_left, Point top_right, Point
         bottom_right, Point bottom_left, Mat object_image);
    ObjectAndLocation(FileNode& node);
    void write(FileStorage& fs);
    void read(FileNode& node);
    Mat& getImage();
    string getName();
    void setName(string new_name);
    string getVerticesString();
    void DrawObject(Mat* display_image, Scalar& colour);
    double getMinimumSideLength();
    double getArea();
    void getVertice(int index, int& x, int& y);
    void setImage(Mat image);   // *** Student should add any initialisation (of
         their images or features; see private data below) they wish into this
        method.
    double compareObjects(ObjectAndLocation* otherObject);  // *** Student
        should write code to compare objects using chosen method.
    bool OverlapsWith(ObjectAndLocation* other_object);
private:
    string object_name;
    Mat image;
    vector<Point2i> vertices;
    // *** Student can add whatever images or features they need to describe the
         object.
};
```

```
class AnnotatedImages;

class ImageWithObjects
{
	friend class AnnotatedImages;
public:
	ImageWithObjects(string passed_filename);
	ImageWithObjects(FileNode& node);
	virtual void LocateAndAddAllObjects(AnnotatedImages& training_images) = 0;
	ObjectAndLocation* addObject(string object_name, int top_left_column, int
	   top_left_row, int top_right_column, int top_right_row,
	        int bottom_right_column, int bottom_right_row, int
	           bottom_left_column, int bottom_left_row, Mat& image);
	void write(FileStorage& fs);
	void read(FileNode& node);
	ObjectAndLocation* getObject(int index);
	void extractAndSetObjectImage(ObjectAndLocation *new_object);
	string ExtractObjectName(string filenamestr);
	void FindBestMatch(ObjectAndLocation* new_object, string& object_name,
	   double& match_value);
protected:
	string filename;
	Mat image;
	vector<ObjectAndLocation> objects;
};

class ImageWithBlueSignObjects : public ImageWithObjects
{
public:
	ImageWithBlueSignObjects(string passed_filename);
	ImageWithBlueSignObjects(FileNode& node);
	void LocateAndAddAllObjects(AnnotatedImages& training_images);  // ***
	   Student needs to develop this routine and add in objects using the
	   addObject method
};

class ConfusionMatrix;

class AnnotatedImages
{
public:
	AnnotatedImages(string directory_name);
	AnnotatedImages();
	void addAnnotatedImage(ImageWithObjects &annotated_image);
	void write(FileStorage& fs);
	void read(FileStorage& fs);
	void read(FileNode& node);
	void read(string filename);
	void LocateAndAddAllObjects(AnnotatedImages& training_images);
	void FindBestMatch(ObjectAndLocation* new_object);
	Mat getImageOfAllObjects(int break_after = 7);
	void CompareObjectsWithGroundTruth(AnnotatedImages& training_images,
	   AnnotatedImages& ground_truth, ConfusionMatrix& results);
	ImageWithObjects* getAnnotatedImage(int index);
	ImageWithObjects* FindAnnotatedImage(string filename_to_find);
public:
	string name;
	vector<ImageWithObjects*> annotated_images;
```

```
};

class ConfusionMatrix
{
public:
        ConfusionMatrix(AnnotatedImages training_images);
        void AddMatch(string ground_truth, string recognised_as, bool duplicate =
           false);
        void AddFalseNegative(string ground_truth);
        void AddFalsePositive(string recognised_as);
        void Print();
private:
        void AddObjectClass(string object_class_name);
        int getObjectClassIndex(string object_class_name);
        vector<string> class_names;
        int confusion_size;
        int** confusion_matrix;
        int false_index;
        int tp, fp, fn;
};

ObjectAndLocation::ObjectAndLocation(string passed_object_name, Point top_left,
   Point top_right, Point bottom_right, Point bottom_left, Mat object_image)
{
        object_name = passed_object_name;
        vertices.push_back(top_left);
        vertices.push_back(top_right);
        vertices.push_back(bottom_right);
        vertices.push_back(bottom_left);
        setImage(object_image);
}
ObjectAndLocation::ObjectAndLocation(FileNode& node)
{
        read(node);
}
void ObjectAndLocation::write(FileStorage& fs)
{
        fs << "{" << "nameStr" << object_name;
        fs << "coordinates" << "[";
        for (int i = 0; i < vertices.size(); ++i)
        {
                fs << "[:" << vertices[i].x << vertices[i].y << "]";
        }
        fs << "]";
        fs << "}";
}
void ObjectAndLocation::read(FileNode& node)
{
        node["nameStr"] >> object_name;
        FileNode data = node["coordinates"];
        for (FileNodeIterator itData = data.begin(); itData != data.end(); ++itData)
        {
                // Read each point
                FileNode pt = *itData;

                Point2i point;
                FileNodeIterator itPt = pt.begin();
                point.x = *itPt; ++itPt;
```

```
                point.y = *itPt;
                vertices.push_back(point);
        }
}
Mat& ObjectAndLocation::getImage()
{
        return image;
}
string ObjectAndLocation::getName()
{
        return object_name;
}
void ObjectAndLocation::setName(string new_name)
{
        object_name.assign(new_name);
}
string ObjectAndLocation::getVerticesString()
{
        string result;
        for (int index = 0; (index < vertices.size()); index++)
                result.append("(" + to_string(vertices[index].x) + " " + to_string(vertices[index].y) + ") ");
        return result;
}
void ObjectAndLocation::DrawObject(Mat* display_image, Scalar& colour)
{
        writeText(*display_image, object_name, vertices[0].y - 8, vertices[0].x + 8, colour, 2.0, 4);
        polylines(*display_image, vertices, true, colour, 8);
}
double ObjectAndLocation::getMinimumSideLength()
{
        double min_distance = DistanceBetweenPoints(vertices[0], vertices[vertices.size() - 1]);
        for (int index = 0; (index < vertices.size() - 1); index++)
        {
                double distance = DistanceBetweenPoints(vertices[index], vertices[index + 1]);
                if (distance < min_distance)
                        min_distance = distance;
        }
        return min_distance;
}
double ObjectAndLocation::getArea()
{
        return contourArea(vertices);
}
void ObjectAndLocation::getVertice(int index, int& x, int& y)
{
        if ((vertices.size() < index) || (index < 0))
                x = y = -1;
        else
        {
                x = vertices[index].x;
                y = vertices[index].y;
        }
}
```

```
ImageWithObjects::ImageWithObjects(string passed_filename)
{
        filename = strdup(passed_filename.c_str());
        cout << "Opening " << filename << endl;
        image = imread(filename, -1);
}
ImageWithObjects::ImageWithObjects(FileNode& node)
{
        read(node);
}
ObjectAndLocation* ImageWithObjects::addObject(string object_name, int
   top_left_column, int top_left_row, int top_right_column, int top_right_row,
        int bottom_right_column, int bottom_right_row, int bottom_left_column, int
           bottom_left_row, Mat& image)
{
        ObjectAndLocation new_object(object_name, Point(top_left_column,
           top_left_row), Point(top_right_column, top_right_row), Point(
           bottom_right_column, bottom_right_row), Point(bottom_left_column,
           bottom_left_row), image);
        objects.push_back(new_object);
        return &(objects[objects.size() - 1]);
}
void ImageWithObjects::write(FileStorage& fs)
{
        fs << "{" << "Filename" << filename << "Objects" << "[";
        for (int index = 0; index < objects.size(); index++)
                objects[index].write(fs);
        fs << "]" << "}";
}
void ImageWithObjects::extractAndSetObjectImage(ObjectAndLocation *new_object)
{
        Mat perspective_warped_image = Mat::zeros(STANDARD_SIGN_WIDTH_AND_HEIGHT,
           STANDARD_SIGN_WIDTH_AND_HEIGHT, image.type());
        Mat perspective_matrix(3, 3, CV_32FC1);
        int x[4], y[4];
        new_object->getVertice(0, x[0], y[0]);
        new_object->getVertice(1, x[1], y[1]);
        new_object->getVertice(2, x[2], y[2]);
        new_object->getVertice(3, x[3], y[3]);
        Point2f source_points[4] = { { ((float)x[0]), ((float)y[0]) },{ ((float)x
           [1]), ((float)y[1]) },{ ((float)x[2]), ((float)y[2]) },{ ((float)x[3]),
           ((float)y[3]) } };
        Point2f destination_points[4] = { { 0.0, 0.0 },{
           STANDARD_SIGN_WIDTH_AND_HEIGHT - 1, 0.0 },{
           STANDARD_SIGN_WIDTH_AND_HEIGHT - 1, STANDARD_SIGN_WIDTH_AND_HEIGHT - 1
           },{ 0.0, STANDARD_SIGN_WIDTH_AND_HEIGHT - 1 } };
        perspective_matrix = getPerspectiveTransform(source_points,
           destination_points);
        warpPerspective(image, perspective_warped_image, perspective_matrix,
           perspective_warped_image.size());
        new_object->setImage(perspective_warped_image);
}
void ImageWithObjects::read(FileNode& node)
{
        filename = (string)node["Filename"];
        image = imread(filename, -1);
        FileNode images_node = node["Objects"];
        if (images_node.type() == FileNode::SEQ)
```

```
	{
		for (FileNodeIterator it = images_node.begin(); it != images_node.
			end(); ++it)
		{
			FileNode current_node = *it;
			ObjectAndLocation *new_object = new ObjectAndLocation(
				current_node);
			extractAndSetObjectImage(new_object);
			objects.push_back(*new_object);
		}
	}
}
ObjectAndLocation* ImageWithObjects::getObject(int index)
{
	if ((index < 0) || (index >= objects.size()))
		return NULL;
	else return &(objects[index]);
}
void ImageWithObjects::FindBestMatch(ObjectAndLocation* new_object, string&
	object_name, double& match_value)
{
	for (int index = 0; (index < objects.size()); index++)
	{
		double temp_match_score = objects[index].compareObjects(new_object);
		if ((temp_match_score > 0.0) && ((match_value < 0.0) || (
			temp_match_score < match_value)))
		{
			object_name = objects[index].getName();
			match_value = temp_match_score;
		}
	}
}

string ImageWithObjects::ExtractObjectName(string filenamestr)
{
	int last_slash = filenamestr.rfind("/");
	int start_of_object_name = (last_slash == std::string::npos) ? 0 :
		last_slash + 1;
	int extension = filenamestr.find(".", start_of_object_name);
	int end_of_filename = (extension == std::string::npos) ? filenamestr.length
		() - 1 : extension - 1;
	int end_of_object_name = filenamestr.find_last_not_of("1234567890",
		end_of_filename);
	end_of_object_name = (end_of_object_name == std::string::npos) ?
		end_of_filename : end_of_object_name;
	string object_name = filenamestr.substr(start_of_object_name,
		end_of_object_name - start_of_object_name + 1);
	return object_name;
}


ImageWithBlueSignObjects::ImageWithBlueSignObjects(string passed_filename) :
	ImageWithObjects(passed_filename)
{
}
ImageWithBlueSignObjects::ImageWithBlueSignObjects(FileNode& node) :
	ImageWithObjects(node)
{
```

```
}


AnnotatedImages::AnnotatedImages(string directory_name)
{
	name = directory_name;
	for (std::experimental::filesystem::directory_iterator next(std::
	    experimental::filesystem::path(directory_name.c_str())), end; next != end
	    ; ++next)
	{
		read(next->path().generic_string());
	}
}
AnnotatedImages::AnnotatedImages()
{
	name = "";
}
void AnnotatedImages::addAnnotatedImage(ImageWithObjects &annotated_image)
{
	annotated_images.push_back(&annotated_image);
}

void AnnotatedImages::write(FileStorage& fs)
{
	fs << "AnnotatedImages";
	fs << "{";
	fs << "name" << name << "ImagesAndObjects" << "[";
	for (int index = 0; index < annotated_images.size(); index++)
		annotated_images[index]->write(fs);
	fs << "]" << "}";
}
void AnnotatedImages::read(FileStorage& fs)
{
	FileNode node = fs.getFirstTopLevelNode();
	read(node);
}
void AnnotatedImages::read(FileNode& node)
{
	name = (string)node["name"];
	FileNode images_node = node["ImagesAndObjects"];
	if (images_node.type() == FileNode::SEQ)
	{
		for (FileNodeIterator it = images_node.begin(); it != images_node.
		    end(); ++it)
		{
			FileNode current_node = *it;
			ImageWithBlueSignObjects* new_image_with_objects = new
			    ImageWithBlueSignObjects(current_node);
			annotated_images.push_back(new_image_with_objects);
		}
	}
}
void AnnotatedImages::read(string filename)
{
	ImageWithBlueSignObjects *new_image_with_objects = new
	    ImageWithBlueSignObjects(filename);
	annotated_images.push_back(new_image_with_objects);
}
```

```
void AnnotatedImages::LocateAndAddAllObjects(AnnotatedImages& training_images)
{
        for (int index = 0; index < annotated_images.size(); index++)
        {
                //if (index == 1) {
                //      break;
                //}
                cout << "Current Image: " << annotated_images[index]->filename <<
                   endl;
                annotated_images[index]->LocateAndAddAllObjects(training_images);
        }
}
void AnnotatedImages::FindBestMatch(ObjectAndLocation* new_object) //Mat&
   perspective_warped_image, string& object_name, double& match_value)
{
        double match_value = -1.0;
        string object_name = "Unknown";
        double temp_best_match = 1000000.0;
        string temp_best_name;
        double temp_second_best_match = 1000000.0;
        string temp_second_best_name;
        for (int index = 0; index < annotated_images.size(); index++)
        {
                annotated_images[index]->FindBestMatch(new_object, object_name,
                   match_value);
                if (match_value < temp_best_match)
                {
                        if (temp_best_name.compare(object_name) != 0)
                        {
                                temp_second_best_match = temp_best_match;
                                temp_second_best_name = temp_best_name;
                        }
                        temp_best_match = match_value;
                        temp_best_name = object_name;
                }
                else if ((match_value != temp_best_match) && (match_value <
                   temp_second_best_match) && (temp_best_name.compare(object_name)
                   != 0))
                {
                        temp_second_best_match = match_value;
                        temp_second_best_name = object_name;
                }
        }
        if (temp_second_best_match / temp_best_match <
           REQUIRED_RATIO_OF_BEST_TO_SECOND_BEST)
                new_object->setName("Unknown");
        else new_object->setName(temp_best_name);
}

Mat AnnotatedImages::getImageOfAllObjects(int break_after)
{
        Mat all_rows_so_far;
        Mat output;
        int count = 0;
        int object_index = 0;
        string blank("");
        for (int index = 0; (index < annotated_images.size()); index++)
        {
```

```
		ObjectAndLocation* current_object = NULL;
		int object_index = 0;
		while ((current_object = (annotated_images[index])->getObject(
			object_index)) != NULL)
		{
			if (count == 0)
			{
				output = JoinSingleImage(current_object->getImage(),
					current_object->getName());
			}
			else if (count % break_after == 0)
			{
				if (count == break_after)
					all_rows_so_far = output;
				else
				{
					Mat temp_rows = JoinImagesVertically(
						all_rows_so_far, blank, output, blank, 0)
						;
					all_rows_so_far = temp_rows.clone();
				}
				output = JoinSingleImage(current_object->getImage(),
					current_object->getName());
			}
			else
			{
				Mat new_output = JoinImagesHorizontally(output,
					blank, current_object->getImage(), current_object
					->getName(), 0);
				output = new_output.clone();
			}
			count++;
			object_index++;
		}
	}
	if (count == 0)
	{
		Mat blank_output(1, 1, CV_8UC3, Scalar(0, 0, 0));
		return blank_output;
	}
	else if (count < break_after)
		return output;
	else {
		Mat temp_rows = JoinImagesVertically(all_rows_so_far, blank, output,
			blank, 0);
		all_rows_so_far = temp_rows.clone();
		return all_rows_so_far;
	}
}

ImageWithObjects* AnnotatedImages::getAnnotatedImage(int index)
{
	if ((index >= 0) && (index < annotated_images.size()))
		return annotated_images[index];
	else return NULL;
}

ImageWithObjects* AnnotatedImages::FindAnnotatedImage(string filename_to_find)
```

```
{
        for (int index = 0; (index < annotated_images.size()); index++)
        {
                if (filename_to_find.compare(annotated_images[index]->filename) ==
                    0)
                        return annotated_images[index];
        }
        return NULL;
}

void MyApplication()
{
        // TRAINING IMAGES HERE
        AnnotatedImages trainingImages;
        FileStorage training_file("BlueSignsTraining.xml", FileStorage::READ);
        if (!training_file.isOpened())
        {
                cout << "Could not open the file: \"" << "BlueSignsTraining.xml" <<
                    "\"" << endl;
        }
        else
        {
                trainingImages.read(training_file);
        }
        training_file.release();
        Mat image_of_all_training_objects = trainingImages.getImageOfAllObjects();
        imshow("All Training Objects", image_of_all_training_objects);
        imwrite("AllTrainingObjectImages.jpg", image_of_all_training_objects);
        char ch = cv::waitKey(1);

        // GROUND TRUTH IMAGES HERE
        AnnotatedImages groundTruthImages;
        FileStorage ground_truth_file("BlueSignsGroundTruth.xml", FileStorage::READ)
            ;
        if (!ground_truth_file.isOpened())
        {
                cout << "Could not open the file: \"" << "BlueSignsGroundTruth.xml"
                    << "\"" << endl;
        }
        else
        {
                groundTruthImages.read(ground_truth_file);
        }
        ground_truth_file.release();
        Mat image_of_all_ground_truth_objects = groundTruthImages.
            getImageOfAllObjects();
        imshow("All Ground Truth Objects", image_of_all_ground_truth_objects);
        imwrite("AllGroundTruthObjectImages.jpg", image_of_all_ground_truth_objects)
            ;
        ch = cv::waitKey(1);

        // IMAGES TO IDENTIFY HERE
        AnnotatedImages unknownImages("Blue Signs/Testing");
        unknownImages.LocateAndAddAllObjects(trainingImages);
        FileStorage unknowns_file("BlueSignsTesting.xml", FileStorage::WRITE);
        if (!unknowns_file.isOpened())
        {
                cout << "Could not open the file: \"" << "BlueSignsTesting.xml" << "
```

```
                \"" << endl;
        }
        else
        {
                unknownImages.write(unknowns_file);
        }
        unknowns_file.release();
        Mat image_of_recognised_objects = unknownImages.getImageOfAllObjects();
        imshow("All Recognised Objects", image_of_recognised_objects);
        imwrite("AllRecognisedObjects.jpg", image_of_recognised_objects);

        // CONFUSION MATRIX HERE
        ConfusionMatrix results(trainingImages);
        unknownImages.CompareObjectsWithGroundTruth(trainingImages,
           groundTruthImages, results);
        results.Print();
}


bool PointInPolygon(Point2i point, vector<Point2i> vertices)
{
        int i, j, nvert = vertices.size();
        bool inside = false;

        for (i = 0, j = nvert - 1; i < nvert; j = i++)
        {
                if ((vertices[i].x == point.x) && (vertices[i].y == point.y))
                        return true;
                if (((vertices[i].y >= point.y) != (vertices[j].y >= point.y)) &&
                        (point.x <= (vertices[j].x - vertices[i].x) * (point.y -
                           vertices[i].y) / (vertices[j].y - vertices[i].y) +
                           vertices[i].x)
                        )
                        inside = !inside;
        }
        return inside;
}

bool ObjectAndLocation::OverlapsWith(ObjectAndLocation* other_object)
{
        double area = contourArea(vertices);
        double other_area = contourArea(other_object->vertices);
        double overlap_area = 0.0;
        int count_points_inside = 0;
        for (int index = 0; (index < vertices.size()); index++)
        {
                if (PointInPolygon(vertices[index], other_object->vertices))
                        count_points_inside++;
        }
        int count_other_points_inside = 0;
        for (int index = 0; (index < other_object->vertices.size()); index++)
        {
                if (PointInPolygon(other_object->vertices[index], vertices))
                        count_other_points_inside++;
        }
        if (count_points_inside == vertices.size())
                overlap_area = area;
        else if (count_other_points_inside == other_object->vertices.size())
```

```
            overlap_area = other_area;
    else if ((count_points_inside == 0) && (count_other_points_inside == 0))
            overlap_area = 0.0;
    else
    {   // There is a partial overlap of the polygons.
            // Find min & max x & y for the current object
            int min_x = vertices[0].x, min_y = vertices[0].y, max_x = vertices[0].x, max_y = vertices[0].y;
            for (int index = 0; (index < vertices.size()); index++)
            {
                    if (min_x > vertices[index].x)
                            min_x = vertices[index].x;
                    else if (max_x < vertices[index].x)
                            max_x = vertices[index].x;
                    if (min_y > vertices[index].y)
                            min_y = vertices[index].y;
                    else if (max_y < vertices[index].y)
                            max_y = vertices[index].y;
            }
            int min_x2 = other_object->vertices[0].x, min_y2 = other_object->vertices[0].y, max_x2 = other_object->vertices[0].x, max_y2 = other_object->vertices[0].y;
            for (int index = 0; (index < other_object->vertices.size()); index++)
            {
                    if (min_x2 > other_object->vertices[index].x)
                            min_x2 = other_object->vertices[index].x;
                    else if (max_x2 < other_object->vertices[index].x)
                            max_x2 = other_object->vertices[index].x;
                    if (min_y2 > other_object->vertices[index].y)
                            min_y2 = other_object->vertices[index].y;
                    else if (max_y2 < other_object->vertices[index].y)
                            max_y2 = other_object->vertices[index].y;
            }
            // We only need the maximum overlapping bounding boxes
            if (min_x < min_x2) min_x = min_x2;
            if (max_x > max_x2) max_x = max_x2;
            if (min_y < min_y2) min_y = min_y2;
            if (max_y > max_y2) max_y = max_y2;
            // For all points
            overlap_area = 0;
            Point2i current_point;
            // Try ever decreasing squares within the overlapping (image aligned) bounding boxes to find the overlapping area.
            bool all_points_inside = false;
            int distance_from_edge = 0;
            for (; ((distance_from_edge < (max_x - min_x + 1) / 2) && (distance_from_edge < (max_y - min_y + 1) / 2) && (!all_points_inside)); distance_from_edge++)
            {
                    all_points_inside = true;
                    for (current_point.x = min_x + distance_from_edge; (current_point.x <= (max_x - distance_from_edge)); current_point.x++)
                            for (current_point.y = min_y + distance_from_edge; (current_point.y <= max_y - distance_from_edge); current_point.y += max_y - 2 * distance_from_edge - min_y)
```

```
                    {
                            if ((PointInPolygon(current_point, vertices)
                               ) && (PointInPolygon(current_point,
                               other_object->vertices)))
                                    overlap_area++;
                            else all_points_inside = false;
                    }
            for (current_point.y = min_y + distance_from_edge + 1; (
               current_point.y <= (max_y - distance_from_edge - 1));
               current_point.y++)
                    for (current_point.x = min_x + distance_from_edge; (
                       current_point.x <= max_x - distance_from_edge);
                       current_point.x += max_x - 2 * distance_from_edge
                        - min_x)
                    {
                            if ((PointInPolygon(current_point, vertices)
                               ) && (PointInPolygon(current_point,
                               other_object->vertices)))
                                    overlap_area++;
                            else all_points_inside = false;
                    }
        }
        if (all_points_inside)
                overlap_area += (max_x - min_x + 1 - 2 * (distance_from_edge
                    + 1)) * (max_y - min_y + 1 - 2 * (distance_from_edge +
                   1));
    }
    double percentage_overlap = (overlap_area*2.0) / (area + other_area);
    return (percentage_overlap >= REQUIRED_OVERLAP);
}



void AnnotatedImages::CompareObjectsWithGroundTruth(AnnotatedImages& training_images
   , AnnotatedImages& ground_truth, ConfusionMatrix& results)
{
    // For every annotated image in ground_truth, find the corresponding image
       in this
    for (int ground_truth_image_index = 0; ground_truth_image_index <
       ground_truth.annotated_images.size(); ground_truth_image_index++)
    {
        ImageWithObjects* current_annotated_ground_truth_image =
           ground_truth.annotated_images[ground_truth_image_index];
        ImageWithObjects* current_annotated_recognition_image =
           FindAnnotatedImage(current_annotated_ground_truth_image->filename
           );

        if (current_annotated_recognition_image != NULL)
        {
            ObjectAndLocation* current_ground_truth_object = NULL;
            int ground_truth_object_index = 0;
            Mat* display_image = NULL;
            if (!current_annotated_recognition_image->image.empty())
            {
                display_image = &(
                   current_annotated_recognition_image->image);
            }
            // For each object in ground_truth.annotated_image
```

```
while ((current_ground_truth_object =
   current_annotated_ground_truth_image->getObject(
   ground_truth_object_index)) != NULL)
{
        if ((current_ground_truth_object->
           getMinimumSideLength() >= MINIMUM_SIGN_SIDE) &&
                (current_ground_truth_object->getArea() >=
                   MINIMUM_SIGN_AREA))
        {
                // Determine the number of overlapping
                   objects (correct & incorrect)
                vector<ObjectAndLocation*>
                   overlapping_correct_objects;
                vector<ObjectAndLocation*>
                   overlapping_incorrect_objects;
                ObjectAndLocation* current_recognised_object
                    = NULL;
                int recognised_object_index = 0;
                // For each object in this.annotated_image
                while ((current_recognised_object =
                   current_annotated_recognition_image->
                   getObject(recognised_object_index)) !=
                   NULL)
                {
                        if (current_recognised_object->
                           getName().compare("Unknown") !=
                           0)
                                if (
                                   current_ground_truth_objec
                                   ->OverlapsWith(
                                   current_recognised_object
                                   ))
                                {
                                        if (
                                           current_ground_tru
                                           ->getName().
                                           compare(
                                           current_recognised
                                           ->getName()) ==
                                           0)
                                                overlapping_c
                                                   .
                                                   push_back
                                                   (
                                                   current_re
                                                   );
                                        else
                                           overlapping_incorr
                                           .push_back(
                                           current_recognised
                                           );
                                }
                        recognised_object_index++;
                }
                if ((overlapping_correct_objects.size() ==
                   0) && (overlapping_incorrect_objects.size
                   () == 0))
                {
```

```
                        if (display_image != NULL)
                        {
                                Scalar colour(0x00, 0x00, 0
                                   xFF);
                                current_ground_truth_object
                                   ->DrawObject(
                                   display_image, colour);
                        }
                        results.AddFalseNegative(
                           current_ground_truth_object->
                           getName());
                        cout <<
                           current_annotated_ground_truth_ima
                           ->filename << ",␣" <<
                           current_ground_truth_object->
                           getName() << ",␣(False␣Negative)␣
                           ,␣" <<
                           current_ground_truth_object->
                           getVerticesString() << endl;
                }
                else {
                        for (int index = 0; (index <
                           overlapping_correct_objects.size
                           ()); index++)
                        {
                                Scalar colour(0x00, 0xFF, 0
                                   x00);
                                results.AddMatch(
                                   current_ground_truth_objec
                                   ->getName(),
                                   overlapping_correct_object
                                   [index]->getName(), (
                                   index > 0));
                                if (index > 0)
                                {
                                        colour[2] = 0xFF;
                                        cout <<
                                           current_annotated_
                                           ->filename << ",␣
                                           " <<
                                           current_ground_tru
                                           ->getName() << ",
                                           ␣(Duplicate)␣,␣"
                                           <<
                                           current_ground_tru
                                           ->
                                           getVerticesString
                                           () << endl;
                                }
                                if (display_image != NULL)
                                        current_ground_truth_
                                           ->DrawObject(
                                           display_image,
                                           colour);
                        }
                        for (int index = 0; (index <
                           overlapping_incorrect_objects.
                           size()); index++)
```

```
                                    {
                                            if (display_image != NULL)
                                            {
                                                    Scalar colour(0xFF,
                                                       0x00, 0xFF);
                                                    overlapping_incorrect
                                                       [index]->
                                                       DrawObject(
                                                       display_image,
                                                       colour);
                                            }
                                            results.AddMatch(
                                               current_ground_truth_objec
                                               ->getName(),
                                               overlapping_incorrect_obje
                                               [index]->getName(), (
                                               index > 0));
                                            cout <<
                                               current_annotated_ground_t
                                               ->filename << ", " <<
                                               current_ground_truth_objec
                                               ->getName() << ", (
                                               Mismatch), " <<
                                               overlapping_incorrect_obje
                                               [index]->getName() << " ,
                                                " <<
                                               current_ground_truth_objec
                                               ->getVerticesString() <<
                                               endl;;
                                    }
                            }
                    }
                    else
                            cout << current_annotated_ground_truth_image
                               ->filename << ", " <<
                               current_ground_truth_object->getName() <<
                                ", (DROPPED_GT) , " <<
                               current_ground_truth_object->
                               getVerticesString() << endl;

                    ground_truth_object_index++;
            }
//          For each object in this.annotated_image
//                                  For each overlapping object
   in ground_truth.annotated_image
//                                          Don't do anything (
   as already done above)
//                          If no overlapping objects.
//                                  Update the confusion table (
   with a False Positive)
            ObjectAndLocation* current_recognised_object = NULL;
            int recognised_object_index = 0;
            // For each object in this.annotated_image
            while ((current_recognised_object =
               current_annotated_recognition_image->getObject(
               recognised_object_index)) != NULL)
            {
                    if ((current_recognised_object->getMinimumSideLength
```

```
        () >= MINIMUM_SIGN_SIDE) &&
            (current_recognised_object->getArea() >=
               MINIMUM_SIGN_AREA))
    {
            // Determine the number of overlapping
               objects (correct & incorrect)
            vector<ObjectAndLocation*>
               overlapping_objects;
            ObjectAndLocation*
               current_ground_truth_object = NULL;
            int ground_truth_object_index = 0;
            // For each object in ground_truth.
               annotated_image
            while ((current_ground_truth_object =
               current_annotated_ground_truth_image->
               getObject(ground_truth_object_index)) !=
               NULL)
            {
                    if (current_ground_truth_object->
                       OverlapsWith(
                       current_recognised_object))
                            overlapping_objects.
                               push_back(
                               current_ground_truth_objec
                               );
                    ground_truth_object_index++;
            }
            if ((overlapping_objects.size() == 0) && (
               current_recognised_object->getName().
               compare("Unknown") != 0))
            {
                    results.AddFalsePositive(
                       current_recognised_object->
                       getName());
                    if (display_image != NULL)
                    {
                            Scalar colour(0x7F, 0x7F, 0
                               xFF);
                            current_recognised_object->
                               DrawObject(display_image,
                                colour);
                    }
                    cout <<
                       current_annotated_recognition_imag
                       ->filename << ", " <<
                       current_recognised_object->
                       getName() << ", (False Positive)
                       , " << current_recognised_object
                       ->getVerticesString() << endl;
            }
    }
    else
            cout << current_annotated_recognition_image
               ->filename << ", " <<
               current_recognised_object->getName() << "
               , (DROPPED) , " <<
               current_recognised_object->
               getVerticesString() << endl;
```

```
                                        recognised_object_index++;
                                }
                                if (display_image != NULL)
                                {
                                        Mat smaller_image;
                                        resize(*display_image, smaller_image, Size(
                                           display_image->cols / 4, display_image->rows / 4)
                                           );
                                        imshow(current_annotated_recognition_image->filename
                                           , smaller_image);
                                        char ch = cv::waitKey(1);
                                        //                              delete display_image
                                           ;
                                }
                        }
                }
}


// Determine object classes from the training_images (vector of strings)
// Create and zero a confusion matrix
ConfusionMatrix::ConfusionMatrix(AnnotatedImages training_images)
{
        // Extract object class names
        ImageWithObjects* current_annnotated_image = NULL;
        int image_index = 0;
        while ((current_annnotated_image = training_images.getAnnotatedImage(
           image_index)) != NULL)
        {
                ObjectAndLocation* current_object = NULL;
                int object_index = 0;
                while ((current_object = current_annnotated_image->getObject(
                   object_index)) != NULL)
                {
                        AddObjectClass(current_object->getName());
                        object_index++;
                }
                image_index++;
        }
        // Create and initialise confusion matrix
        confusion_size = class_names.size() + 1;
        confusion_matrix = new int*[confusion_size];
        for (int index = 0; (index < confusion_size); index++)
        {
                confusion_matrix[index] = new int[confusion_size];
                for (int index2 = 0; (index2 < confusion_size); index2++)
                        confusion_matrix[index][index2] = 0;
        }
        false_index = confusion_size - 1;
}
void ConfusionMatrix::AddObjectClass(string object_class_name)
{
        int index = getObjectClassIndex(object_class_name);
        if (index == -1)
                class_names.push_back(object_class_name);
        tp = fp = fn = 0;
}
int ConfusionMatrix::getObjectClassIndex(string object_class_name)
```

```
{
        int index = 0;
        for (; (index < class_names.size()) && (object_class_name.compare(class_names[index]) != 0); index++)
                ;
        if (index < class_names.size())
                return index;
        else return -1;
}
void ConfusionMatrix::AddMatch(string ground_truth, string recognised_as, bool duplicate)
{
        if ((ground_truth.compare(recognised_as) == 0) && (duplicate))
                AddFalsePositive(recognised_as);
        else
        {
                confusion_matrix[getObjectClassIndex(ground_truth)][getObjectClassIndex(recognised_as)]++;
                if (ground_truth.compare(recognised_as) == 0)
                        tp++;
                else {
                        fp++;
                        fn++;
                }
        }
}
void ConfusionMatrix::AddFalseNegative(string ground_truth)
{
        fn++;
        confusion_matrix[getObjectClassIndex(ground_truth)][false_index]++;
}
void ConfusionMatrix::AddFalsePositive(string recognised_as)
{
        fp++;
        confusion_matrix[false_index][getObjectClassIndex(recognised_as)]++;
}
void ConfusionMatrix::Print()
{
        cout << ",,,Recognised␣as:" << endl << ",,";
        for (int recognised_as_index = 0; recognised_as_index < confusion_size; recognised_as_index++)
                if (recognised_as_index < confusion_size - 1)
                        cout << class_names[recognised_as_index] << ",";
                else cout << "False␣Negative,";
        cout << endl;
        for (int ground_truth_index = 0; (ground_truth_index <= class_names.size()); ground_truth_index++)
        {
                if (ground_truth_index < confusion_size - 1)
                        cout << "Ground␣Truth," << class_names[ground_truth_index] << ",";
                else cout << "Ground␣Truth,False␣Positive,";
                for (int recognised_as_index = 0; recognised_as_index < confusion_size; recognised_as_index++)
                        cout << confusion_matrix[ground_truth_index][recognised_as_index] << ",";
                cout << endl;
        }
```

```
        double precision = ((double)tp) / ((double)(tp + fp));
        double recall = ((double)tp) / ((double)(tp + fn));
        double f1 = 2.0*precision*recall / (precision + recall);
        cout << endl << "Precision = " << precision << endl << "Recall = " << recall
            << endl << "F1 = " << f1 << endl;
}

// A simple function that takes in an image and returns a sharpened version of that
   image.
Mat my_sharpen_function(Mat original_image) {
        Mat image_32bit, sharpened_image, laplacian;
        Mat image1 = original_image.clone();
        image1.convertTo(image_32bit, CV_32F);
        Laplacian(image1, laplacian, CV_32F, 3);
        Mat sharpened_image_32bit = image_32bit - 0.3*laplacian;
        sharpened_image_32bit.convertTo(sharpened_image, CV_8U);
        return sharpened_image;
}

// Function takes in a contour with 5 vertices and returns an array of contours with
    4 vertices,
// whose area are max of all contours with 4 vertices out of those 5..
// I will try to explain this more in the report, I know it is a little weird.
void my_smooth_contour_function(Mat hull, Mat *hulls) {
        Point top_left;
        Point top_right;
        Point bottom_left;
        Point bottom_right;
        Mat max_area;
        int count = 0;
        Mat temp_hulls[4];
        // I know this is very inefficient, if I had more time I would have found a
           more optimal solution.
        // Since N will only ever be 5 it's not a huge deal..
        // Basically we're finding every 4 sided 'square like' contour, and putting
           it into an array.
        for (int i = 0; i < hull.size[0]; i++) {
                top_left = Point(hull.at<int>(i, 0), hull.at<int>(i, 1));
                for (int j = 0; j < hull.size[0]; j++) {
                        top_right = Point(hull.at<int>(j, 0), hull.at<int>(j, 1));
                        if (top_right.x > top_left.x) {
                                for (int x = 0; x < hull.size[0]; x++) {
                                        bottom_left = Point(hull.at<int>(x, 0), hull
                                            .at<int>(x, 1));
                                        if (bottom_left.y > top_left.y &&
                                            bottom_left.y > top_right.y &&
                                            bottom_left.x < top_right.x) {
                                                for (int y = 0; y < hull.size[0]; y
                                                    ++) {
                                                        bottom_right = Point(hull.at
                                                            <int>(y, 0), hull.at<int
                                                            >(y, 1));
                                                        if (bottom_right.y >
                                                            top_left.y &&
                                                            bottom_right.y >
                                                            top_right.y &&
                                                            bottom_left.x <
                                                            bottom_right.x &&
```

```
                                                                top_left.x < bottom_right
                                                                .x) {
                                                                      vector<Point> data;
                                                                      data.push_back(
                                                                         top_right);
                                                                      data.push_back(
                                                                         top_left);
                                                                      data.push_back(
                                                                         bottom_left);
                                                                      data.push_back(
                                                                         bottom_right);
                                                                      Mat temp = Mat(data)
                                                                         .clone();
                                                                      Rect tempRect =
                                                                         boundingRect(temp
                                                                         );
                                                                      Rect max_rect =
                                                                         boundingRect(
                                                                         max_area);
                                                                      if (tempRect.area()
                                                                        >= max_rect.area
                                                                         ()) {
                                                                              temp_hulls[
                                                                                 count] =
                                                                                 temp;
                                                                              count++;
                                                                      }
                                                              }
                                                      }
                                              }
                                      }
                              }
                      }
              }
        // Here we find the max area of all of these contours..
        int max = 0;
        count = 0;
        for (int i = 0; i < 4; i++) {
                Rect myRect = boundingRect(temp_hulls[i]);
                if (myRect.area() > max)
                        max = myRect.area();
        }
        // And put all contours whose area is the same as the max into our returned
           array
        // (A lot of contours did have the same max area)
        for (int i = 0; i < 4; i++) {
                Rect myRect = boundingRect(temp_hulls[i]);
                if (myRect.area() == max) {
                        hulls[count] = temp_hulls[i];
                        count++;
                }
        }

}

// A function that crops a sign and transforms it so we have a front on view of it.
// myRect: Rect of the contour to crop the image..
// image : Mat of the image we want to crop from.
```

```
// hull  : hull of the contour we want to crop.. (looking back we don't need to rect
    object..)
// factor: In case we don't multiply before we enter the function..
// Returns the front on view of the sign..
Mat my_crop_and_distort_function(Rect myRect, Mat image, Mat hull, double factor) {
        // Getting Co-ordinate of signs on larger image and cropping that out...
        myRect.height *= factor;
        myRect.width *= factor;
        myRect.x *= factor;
        myRect.y *= factor;
        Mat cropped = image(myRect);

        // Finding the min_x and min_y of the cropped image (co-ordinates of the
            cropped image on the actual image...)
        int min_x = image.cols;
        int min_y = image.rows;
        for (int i = 0; i < 4; i++) {
                if ((int)(hull.at<int>(i, 0) * factor) < min_x)
                        min_x = (int)(hull.at<int>(i, 0) * factor);
                if ((int)(hull.at<int>(i, 1) * factor) < min_y)
                        min_y = (int)(hull.at<int>(i, 1) * factor);
        }

        // Creating a Point2f vector of the relative points of rectangle on the
            cropped image...
        Point2f src_v[4];
        src_v[0] = Point((int)(hull.at<int>(3, 0) * factor) - min_x, (int)(hull.at<
            int>(3, 1) * factor) - min_y);
        src_v[1] = Point((int)(hull.at<int>(2, 0) * factor) - min_x, (int)(hull.at<
            int>(2, 1) * factor) - min_y);
        src_v[2] = Point((int)(hull.at<int>(0, 0) * factor) - min_x, (int)(hull.at<
            int>(0, 1) * factor) - min_y);
        src_v[3] = Point((int)(hull.at<int>(1, 0) * factor) - min_x, (int)(hull.at<
            int>(1, 1) * factor) - min_y);

        // Creating a Point2f vector of the corners of the cropped image...
        Point2f dest_v[4];
        dest_v[0] = Point(0, 0);
        dest_v[1] = Point(cropped.rows - 1, 0);
        dest_v[2] = Point(0, cropped.cols - 1);
        dest_v[3] = Point(cropped.rows - 1, cropped.cols - 1);

        // Transforming the cropped image so we have a face on view of our sign...
        Mat perspective;
        Mat warped;
        perspective = getPerspectiveTransform(src_v, dest_v);
        warpPerspective(cropped, warped, perspective, Size(cropped.rows, cropped.
            cols));
        resize(warped, warped, Size(200, 200));
        flip(warped, warped, 1);
        return warped;
}

// Takes in a transformed sign image and returns the best matched training image
    using template matching.
Mat my_template_matching_function(AnnotatedImages& training_images, Mat warped,
    double *max_score, String *best_match_name) {
        // Template matching... (I do template matching for every image, find the
```

```
    average pixel value of the correlation matrix and use the highest average
     as best matching image.)
ImageWithObjects* current_annnotated_image = NULL;
int image_index = 0;
int best_match_index = 0;
Mat correct_orientation;
while ((current_annnotated_image = training_images.getAnnotatedImage(
   image_index)) != NULL)
{
        ObjectAndLocation* current_object = NULL;
        int object_index = 0;
        while ((current_object = current_annnotated_image->getObject(
           object_index)) != NULL)
        {
                // The reason for this loop is because some of my signs were
                    detected at a different orientation.
                // I rotate the image 4 times at 90 degrees and find the
                   best match of all of those..
                // I know this is a bit weird and probably introduced a few
                   false positives (e.g.) the bike being recognised as 1.
                for (int i = 0; i < 4; i++) {
                        Mat rotated_warped = warped.clone();
                        rotate(rotated_warped, rotated_warped, i);
                        Mat display_image, correlation_image;
                        rotated_warped.copyTo(display_image);
                        double min_correlation, max_correlation;
                        Mat matched_template_map;
                        correlation_image.create(200, 200, CV_32FC1);
                        // TM_CCORR_NORMED seemed to work better than any
                           other method..
                        // TM_CCOEF_NORMED actually consistently gave me 1
                           precision, but I would lose a lot of recall score
                           .
                        matchTemplate(rotated_warped, current_object->
                           getImage(), correlation_image, cv::
                           TM_CCORR_NORMED);
                        minMaxLoc(correlation_image, &min_correlation, &
                           max_correlation);
                        FindLocalMaxima(correlation_image,
                           matched_template_map, max_correlation*0.99);
                        Mat matched_template_display1;
                        cvtColor(matched_template_map,
                           matched_template_display1, COLOR_GRAY2BGR);
                        Mat correlation_window1 =
                           convert_32bit_image_for_display(correlation_image
                           , 0.0);
                        Scalar avg = mean(correlation_image);
                        // If this image is a better match, replace the old
                           one..
                        if (avg[0] > *max_score) {
                                *max_score = avg[0];
                                best_match_index = image_index;
                                correct_orientation = rotated_warped;
                        }
                }
                object_index++;
        }
        image_index++;
```

```
	}
	// Again through trial and error, 0.835 seemed to get the best results for
	   me.
	if (*max_score > 0.835)
		*best_match_name = training_images.getAnnotatedImage(
		   best_match_index)->getObject(0)->getName();
	else
		*best_match_name = "";
	return correct_orientation;
}

// Takes in two contours and returns whether or not they're overlapping (duplicates)
// I made an arbitrary threshold through trial and error, and this seemed to work
   best.
// Basically just checks if one point of the sign is within thresh pixel radius of
   any other point of the other sign.
// Had to implement like this since orientation of contour isn't actually known.
int my_remove_duplicates_function(Mat hull1, Mat hull2) {
	Rect myRect = boundingRect(hull1);
	int threshold = (int)pow((myRect.width / 100) + 3.25, 1.75);
	if (abs(hull1.at<int>(3, 0) - hull2.at<int>(0, 0)) < threshold && abs(hull1.
	   at<int>(0, 1) - hull2.at<int>(0, 1)) < threshold ||
		abs(hull1.at<int>(3, 0) - hull2.at<int>(1, 0)) < threshold && abs(
		   hull1.at<int>(0, 1) - hull2.at<int>(1, 1)) < threshold ||
		abs(hull1.at<int>(3, 0) - hull2.at<int>(2, 0)) < threshold && abs(
		   hull1.at<int>(0, 1) - hull2.at<int>(2, 1)) < threshold ||
		abs(hull1.at<int>(3, 0) - hull2.at<int>(3, 0)) < threshold && abs(
		   hull1.at<int>(0, 1) - hull2.at<int>(3, 1)) < threshold) {
		return 1;
	}
	return 0;
}

// I tried implementing SVMs using a number of different features for quite some
   time with no luck. I was getting weird values when predicting.
// It would actually be a similar value that you would get from your example.
// I just wasn't sure how to implement it for multiple classes and not just binary.
// I guess this function can just be ignored since it isn't called.
void my_SupportVectorMachineDemo_function(Mat class_samples[], Mat testing_img)
{
	Ptr<ml::SVM> svm;
	int num_images = 30;
	int img_area = class_samples[0].cols * class_samples[0].rows;
	//Mat training_mat(num_images, img_area, CV_32FC1);
	float labels[30];
	float training_data[30][1];
	int number_of_samples = 0;
	for (int x = 0; x < 30; x++) {
		//cout << class_samples[x].size << endl;
		//class_samples[x] = class_samples[x].reshape(1, 1);
		//cout << class_samples[x].size << endl;
		//int ii = 0;
		//for (int i = 0; i < class_samples[x].rows; i++) {
		//	for (int j = 0; j < class_samples[x].cols; j++) {
		//		training_mat.at<float>(x, ii++) = class_samples[x].
		   at<uchar>(i, j);
		//	}
		//}
```

```
Mat gray_image, binary_image;
cvtColor(class_samples[x], gray_image, COLOR_BGR2GRAY);
int padding = 10;
cv::Mat crop = cv::Mat(gray_image, cv::Rect(padding, padding,
   gray_image.cols - 2 * padding, gray_image.rows - 2 * padding));
threshold(crop, binary_image, 110, 255, THRESH_BINARY);
vector<vector<Point>> contours;
vector<Vec4i> hierarchy;
findContours(binary_image, contours, hierarchy, cv::RETR_TREE, cv::
   CHAIN_APPROX_NONE);
Mat contours_image = Mat::zeros(binary_image.size(), CV_8UC3);
contours_image = Scalar(255, 255, 255);
// Do some processing on all contours (objects and holes!)
vector<RotatedRect> min_bounding_rectangle(contours.size());
vector<vector<Point>> hulls(contours.size());
vector<vector<int>> hull_indices(contours.size());
vector<vector<Vec4i>> convexity_defects(contours.size());
vector<Moments> contour_moments(contours.size());
for (int contour_number = 0; (contour_number < (int)contours.size())
   ; contour_number++)
{
        if (contours[contour_number].size() > 10)
        {
                min_bounding_rectangle[contour_number] = minAreaRect
                   (contours[contour_number]);
                convexHull(contours[contour_number], hulls[
                   contour_number]);
                convexHull(contours[contour_number], hull_indices[
                   contour_number]);
                convexityDefects(contours[contour_number],
                   hull_indices[contour_number], convexity_defects[
                   contour_number]);
                contour_moments[contour_number] = moments(contours[
                   contour_number]);
        }
}
for (int contour_number = 0; (contour_number >= 0); contour_number =
     hierarchy[contour_number][0])
{
        if (contours[contour_number].size() > 10)
        {
                Scalar colour(rand() & 0x7F, rand() & 0x7F, rand() &
                    0x7F);
                drawContours(contours_image, contours,
                   contour_number, colour, cv::FILLED, 8, hierarchy)
                   ;
                char output[500];
                double area = contourArea(contours[contour_number])
                   + contours[contour_number].size() / 2 + 1;
                // Process any holes (removing the area from the are
                    of the enclosing contour)
                for (int hole_number = hierarchy[contour_number][2];
                    (hole_number >= 0); hole_number = hierarchy[
                   hole_number][0])
                {
                        area -= (contourArea(contours[hole_number])
                           - contours[hole_number].size() / 2 + 1);
                        Scalar colour(rand() & 0x7F, rand() & 0x7F,
```

```
                rand() & 0x7F);
        drawContours(contours_image, contours,
                hole_number, colour, cv::FILLED, 8,
                hierarchy);
        sprintf(output, "Area=%.0f", contourArea(
                contours[hole_number]) - contours[
                hole_number].size() / 2 + 1);
        Point location(contours[hole_number][0].x +
                20, contours[hole_number][0].y + 5);
        putText(contours_image, output, location,
                FONT_HERSHEY_SIMPLEX, 0.4, colour);
}
// Draw the minimum bounding rectangle
Point2f bounding_rect_points[4];
min_bounding_rectangle[contour_number].points(
        bounding_rect_points);
line(contours_image, bounding_rect_points[0],
        bounding_rect_points[1], Scalar(0, 0, 127));
line(contours_image, bounding_rect_points[1],
        bounding_rect_points[2], Scalar(0, 0, 127));
line(contours_image, bounding_rect_points[2],
        bounding_rect_points[3], Scalar(0, 0, 127));
line(contours_image, bounding_rect_points[3],
        bounding_rect_points[0], Scalar(0, 0, 127));
float bounding_rectangle_area =
        min_bounding_rectangle[contour_number].size.area
        ();
// Draw the convex hull
drawContours(contours_image, hulls, contour_number,
        Scalar(127, 0, 127));
// Highlight any convexities
int largest_convexity_depth = 0;
for (int convexity_index = 0; convexity_index < (int
        )convexity_defects[contour_number].size();
        convexity_index++)
{
        if (convexity_defects[contour_number][
                convexity_index][3] >
                largest_convexity_depth)
                largest_convexity_depth =
                        convexity_defects[contour_number
                        ][convexity_index][3];
        if (convexity_defects[contour_number][
                convexity_index][3] > 256 * 2)
        {
                line(contours_image, contours[
                        contour_number][convexity_defects
                        [contour_number][convexity_index
                        ][0]], contours[contour_number][
                        convexity_defects[contour_number
                        ][convexity_index][2]], Scalar(0,
                         0, 255));
                line(contours_image, contours[
                        contour_number][convexity_defects
                        [contour_number][convexity_index
                        ][1]], contours[contour_number][
                        convexity_defects[contour_number
                        ][convexity_index][2]], Scalar(0,
```

```
                                                    0, 255));
                                        }
                                }
                                // Compute moments and a measure of the deepest
                                    convexity
                                //double hu_moments[7];
                                //HuMoments(contour_moments[contour_number],
                                    hu_moments);
                                //double diameter = ((double)contours[contour_number
                                    ].size()) / PI;
                                ////double convexity_depth = ((double)
                                    largest_convexity_depth)/256.0;
                                //double convex_measure = largest_convexity_depth /
                                    diameter;
                                //int class_id = x;
                                //float feature[2] = { (float)convex_measure*((float
                                    )30), (float)hu_moments[0] * ((float)511) };
                                //if (feature[0] > ((float)511)) feature[0] = ((
                                    float)511);
                                //if (feature[1] > ((float)511)) feature[1] = ((
                                    float)511);
                                //training_data[number_of_samples][0] = feature[0];
                                //training_data[number_of_samples][1] = feature[1];
                                //number_of_samples++;
                                training_data[x][0] = contours[contour_number].size
                                    ();
                                //sprintf(output, "Class=%s, Features %.2f, %.2f", x
                                    , feature[0] / ((float)511), feature[1] / ((float
                                    )511));
                                //Point location(contours[contour_number][0].x - 40,
                                     contours[contour_number][0].y - 3);
                                //putText(contours_image, output, location,
                                    FONT_HERSHEY_SIMPLEX, 0.4, colour);
                        }
                }
                if (x == 0)
                        labels[x] = 0;
                else if (x == 1 || x == 2)
                        labels[x] = 1;
                else if (x == 3 || x == 4)
                        labels[x] = 2;
                else if (x >= 5 && x <= 9)
                        labels[x] = 3;
                else if (x == 10 || x == 11)
                        labels[x] = 4;
                else if (x == 12 || x == 13)
                        labels[x] = 5;
                else if (x == 14 || x == 15)
                        labels[x] = 6;
                else if (x == 16 || x == 17)
                        labels[x] = 7;
                else if (x == 18 || x == 19)
                        labels[x] = 8;
                else if (x >= 20 && x <= 23)
                        labels[x] = 9;
                else if (x >= 24 && x <= 26)
                        labels[x] = 10;
                else if (x >= 27 && x <= 29)
```

```
                        labels[x] = 11;
        }
        svm = ml::SVM::create();
        svm->setType(ml::SVM::C_SVC);
        svm->setKernel(ml::SVM::RBF);
        Mat labelsMat(30, 1, CV_32SC1, labels);
        Mat trainingDataMat(30, 1, CV_32FC1, training_data);
        Ptr<ml::TrainData> tData = ml::TrainData::create(trainingDataMat, ml::
           SampleTypes::ROW_SAMPLE, labelsMat);
        svm->train(tData);
}

// I wasn't sure about this function so I left it blank.
void ObjectAndLocation::setImage(Mat object_image)
{
        image = object_image.clone();
        //imshow("Testing", image);
        // *** Student should add any initialisation (of their images or features;
           see private data below) they wish into this method.
}



void ImageWithBlueSignObjects::LocateAndAddAllObjects(AnnotatedImages&
   training_images)
{
        // *** Student needs to develop this routine and add in objects using the
           addObject method

        // I found I got best results from using two resizing of the images.
        // One to get the bigger signs and another to get the smaller signs.
        // This seems weird to me, I would have thought using the full sized image
           would have been best, since it has the most information within it, but
           didn't seem to work for me.
        // I was mainly going for the highest score I could get and that's why I
           stuck with this.
        Mat *display_image = &(this->image);
        Mat smaller_image;
        resize(*display_image, smaller_image, Size(display_image->cols / 4,
           display_image->rows / 4));
        Mat original_image = this->image.clone();
        Mat original_image_3;
        Mat original_image_4;
        resize(original_image, original_image_3, Size(original_image.cols / 3,
           original_image.rows / 3));
        resize(original_image, original_image_4, Size(original_image.cols / 4,
           original_image.rows / 4));

        // Sharpening the image seemed to give good results.
        // Gave me nice definitive edges for my edge detection.
        Mat sharpened_image_3 = my_sharpen_function(original_image_3);
        Mat sharpened_image_4 = my_sharpen_function(original_image_4);

        // Standard grayscale conversion
        Mat gray_image_4;
        Mat gray_image_3;
        cvtColor(sharpened_image_4, gray_image_4, COLOR_BGR2GRAY);
        cvtColor(sharpened_image_3, gray_image_3, COLOR_BGR2GRAY);
```

```
// Adaptive thresholding worked great, meant I could ignore all of the
   lighting issues within the images.
Mat thresh_4;
Mat thresh_3;
adaptiveThreshold(gray_image_4, thresh_4, 255, ADAPTIVE_THRESH_GAUSSIAN_C,
   THRESH_BINARY_INV, 3, 10);
adaptiveThreshold(gray_image_3, thresh_3, 255, ADAPTIVE_THRESH_GAUSSIAN_C,
   THRESH_BINARY_INV, 3, 25);

// Finding all contours in the image (We want the external contours so we
   can ignore contours within the signs..)
vector<vector<Point>> contours_4;
vector<vector<Point>> contours_3;
vector<vector<Point>> tempcontours_4;
vector<vector<Point>> tempcontours_3;
findContours(thresh_4, contours_4, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE);
findContours(thresh_3, contours_3, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE);
vector<Vec4i> hierarchy;

vector<Mat> all_hulls;
// Finding all contours in image/4
for (int i = 0; i < contours_4.size(); i++)
{
        Mat hull;
        // Ignoring small contours
        if (contourArea(contours_4[i]) > 1000) {
                convexHull(contours_4[i], hull);
                approxPolyDP(hull, hull, RETR_LIST, CHAIN_APPROX_SIMPLE);
                //If we have 4 vertices we have a quadrilateral.. (Some of
                   my contours for signs have 5 and not sure how to fix this
                   ..)
                if (hull.size[0] >= 4 && hull.size[0] <= 5) {
                        // If we don't have 4 distinct vertices we need to
                           smooth the contour...
                        if (hull.size[0] != 4) {
                                Mat hulls[4];
                                my_smooth_contour_function(hull, hulls);
                                for (int i = 0; i < 4; i++)
                                        all_hulls.push_back(hulls[i] * 4);
                        }
                        else
                                all_hulls.push_back(hull * 4);
                }
        }
}
// Finding all contours in image/3
for (int i = 0; i < contours_3.size(); i++)
{
        Mat hull;
        // Ignoring small contours
        if (contourArea(contours_3[i]) > 1000) {
                convexHull(contours_3[i], hull);
                approxPolyDP(hull, hull, RETR_LIST, CHAIN_APPROX_SIMPLE);
                //If we have 4 vertices we have a quadrilateral.. (Some of
                   my contours for signs have 5 and not sure how to fix this
                   ..)
                if (hull.size[0] >= 4 && hull.size[0] <= 5) {
```

```
                        // If we don't have 4 distinct vertices we need to
                           smooth the contour...
                        if (hull.size[0] != 4) {
                                Mat hulls[4];
                                my_smooth_contour_function(hull, hulls);
                                for (int i = 0; i < 4; i++)
                                        all_hulls.push_back(hulls[i] * 3);
                        }
                        all_hulls.push_back(hull * 3);
                }
        }
}

vector<String> best_matches;
vector<double> scores;
vector<Mat> new_hulls;
vector<Mat> warped_imgs;
// For all the hulls, crop and transform them if they're kind of like a
   square..
// The do template matching on them..
for (int i = 0; i < all_hulls.size(); i++) {
        Rect myRect = boundingRect(all_hulls[i]);
        double ar = myRect.width / double(myRect.height);
        // Checking that the quadrilateral is "square like" and not a
           rectangle etc..
        if (ar >= 0.67 && ar <= 1.33) {
                Mat warped = my_crop_and_distort_function(myRect, this->
                   image, all_hulls[i], 1);
                double avg_max = 0;
                String best_match;
                warped = my_template_matching_function(training_images,
                   warped, &avg_max, &best_match);
                warped_imgs.push_back(warped);
                best_matches.push_back(best_match);
                scores.push_back(avg_max);
                new_hulls.push_back(all_hulls[i]);
        }
}

// If we have a duplicate identification, delete the one that got a lower
   score on template matching.
for (int i = 0; i < new_hulls.size(); i++) {
        for (int j = 0; j < new_hulls.size(); j++) {
                if (i != j) {
                        if (my_remove_duplicates_function(new_hulls[i],
                           new_hulls[j]) == 1) {
                                // These two should be vaguely similar..
                                cout << "Removing duplicate..." << endl;
                                if (scores[i] > scores[j]) {
                                        new_hulls.erase(new_hulls.begin() +
                                           j);
                                        best_matches.erase(best_matches.
                                           begin() + j);
                                        scores.erase(scores.begin() + j);
                                        warped_imgs.erase(warped_imgs.begin
                                           () + j);
                                }
                                else {
```

```
                                        new_hulls.erase(new_hulls.begin() +
                                            i);
                                        best_matches.erase(best_matches.
                                            begin() + i);
                                        scores.erase(scores.begin() + i);
                                        warped_imgs.erase(warped_imgs.begin
                                            () + i);
                                }
                                break;
                        }
                }
        }
}

// Finally, add the object!
for (int i = 0; i < new_hulls.size(); i++) {
        if (best_matches[i] != "") {
                addObject(best_matches[i], new_hulls[i].at<int>(3, 0),
                    new_hulls[i].at<int>(3, 1),
                        new_hulls[i].at<int>(2, 0), new_hulls[i].at<int>(2,
                            1),
                        new_hulls[i].at<int>(1, 0), new_hulls[i].at<int>(1,
                            1),
                        new_hulls[i].at<int>(0, 0), new_hulls[i].at<int>(0,
                            1), warped_imgs[i]);
        }
}

// ---------------------------------------------------------------- END OF PROGRAM
    ----------------------------------------------------------------------
// ---------------------------------------------------- BELOW ARE OTHER METHODS i'VE
    TRIED --------------------------------------------------------

// Creating array of training images for svm...
/*Mat training_images_arr[30];
ImageWithObjects* current_annnotated_image = NULL;
int image_index = 0;
while ((current_annnotated_image = training_images.getAnnotatedImage(
    image_index)) != NULL)
{
        ObjectAndLocation* current_object = NULL;
        int object_index = 0;
        while ((current_object = current_annnotated_image->getObject(
            object_index)) != NULL)
        {
                training_images_arr[image_index] = current_object->getImage
                    ();
                object_index++;
        }
        image_index++;
}*/

//Mat *unknown_imgs = &warped_imgs[0];
//SupportVectorMachineDemo(training_images_arr, warped_imgs.at(0));

//imshow(this->filename+"thresh", thresh);
//resize(drawing, drawing, Size(drawing.cols / 4, drawing.rows /4));
//imshow(this->filename, drawing);
```

```
/*
// LAB Colour space supposed to be less sensitive to light...
//Mat LABImage;
//cvtColor(smaller_image, LABImage, COLOR_RGB2Lab);
//imshow(this->filename, LABImage);

// Smoothen image...
Mat smoothed;
GaussianBlur(smaller_image, smoothed, Size(3,3), 0);
imshow(this->filename + "blur", smoothed);

// Equalizing images...
std::vector<cv::Mat> input_planes(3);
Mat processed_image, original_image;
original_image = sharpened_image;
Mat hls_image;
cvtColor(original_image, hls_image, COLOR_BGR2HLS);
//Mat LABImage;
//cvtColor(smaller_image, LABImage, COLOR_RGB2Lab);
split(hls_image, input_planes);
equalizeHist(input_planes[1], input_planes[1]);
merge(input_planes, hls_image);
cvtColor(hls_image, processed_image, COLOR_HLS2BGR);
imshow(this->filename, smaller_image);

*/
// K means clustering...
//Mat kmeans = kmeans_clustering(sharpened_image, 15, 3);
//imshow(this->filename, kmeans);

// Back projection to locate signs... kind of works for some images
//ImageWithObjects* current_annnotated_image = NULL;
//int image_index = 0;
//Mat imgs[33];
//while ((current_annnotated_image = training_images.getAnnotatedImage(
    image_index)) != NULL)
//{
//        ObjectAndLocation* current_object = NULL;
//        int object_index = 0;
//        while ((current_object = current_annnotated_image->getObject(
    object_index)) != NULL)
//        {
//                //cout << image_index << endl;
//                imgs[image_index] = current_object->getImage();
//                object_index++;
//        }
//        image_index++;
//}
//Mat back_proj_prob_img = BackProjection(kmeans, imgs, image_index);
//imshow(this->filename, back_proj_prob_img);

// Canny Edge
//Canny(gray_image, thresh, 450, 1350, 3);

// Dilate & Erode
//Mat dilated;
//dilate(thresh, dilated, Mat(), Point(-1, -1), 1);
```

```
        //Mat eroded;
        //erode(thresh, eroded, Mat(), Point(-1, -1), 2);

        /*
        // harris corners works very well for finding corners of signs..
        Mat image1_gray, image2_gray, image3_gray;
        cvtColor(smaller_image, image1_gray, COLOR_BGR2GRAY);
        Mat harris_cornerness, possible_harris_corners, harris_corners;
        cornerHarris(image1_gray, harris_cornerness, 3, 3, 0.02);
        // V3.0.0 change
        Ptr<FeatureDetector> harris_feature_detector = GFTTDetector::create(1000,
            0.01, 10, 3, true);
        //GoodFeaturesToTrackDetector harris_detector( 1000, 0.01, 10, 3, true );
        vector<KeyPoint> keypoints;
        // V3.0.0 change
        harris_feature_detector->detect(image1_gray, keypoints);
        //harris_detector.detect( image1_gray,keypoints );
        Mat harris_corners_image;
        drawKeypoints(smaller_image, keypoints, harris_corners_image, Scalar(0, 0,
            255));
        imshow(this->filename, harris_corners_image);

        */
        /*
        // Hough tranform for (full) line detection
        vector<Vec2f> hough_lines;
        HoughLines(eroded, hough_lines, 1, PI / 200.0, 100);
        Mat hough_lines_image = smaller_image.clone();
        DrawLines2(hough_lines_image, hough_lines);
        //imshow(this->filename, hough_lines_image);

        // Probabilistic Hough transform for line segments
        vector<Vec4i> hough_line_segments;
        HoughLinesP(canny_edge_image, hough_line_segments, 1.0, PI / 200.0, 20, 20,
            5);
        Mat hough_line_segments_image = Mat::zeros(canny_edge_image.size(), CV_8UC3)
            ;
        DrawLines2(hough_line_segments_image, hough_line_segments);
        Mat hough_img = JoinImagesHorizontally(smaller_image, "Original Image",
            hough_line_segments_image, "Probabilistic Hough (for line segments)", 4);
        imshow(this->filename, hough_img);

        */

}

// I implemented my own function 'my_template_matching_function'.
// Hope this is okay.
#define BAD_MATCHING_VALUE 1000000000.0;
double ObjectAndLocation::compareObjects(ObjectAndLocation* otherObject)
{
        // *** Student should write code to compare objects using chosen method.
        // Please bear in mind that ImageWithObjects::FindBestMatch assumes that the
             lower the value the better.  Feel free to change this.

        // I implemented my own function called 'my_template_matching_function'.

        return BAD_MATCHING_VALUE;
```

}