```
#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/ml.hpp>
#include <map>
#include <string>

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

vector<std::string> Classes = {
  "Coffee",
  "Disabled",
  "Escalator",
  "Exit",
  "Gents",
  "Information",
  "Ladies",
  "Lift",
  "One",
  "Stairs",
  "TicketDesk",
  "Two"
};

void showImage(string name, Mat image)
{
  namedWindow(name, WINDOW_NORMAL);
  imshow(name, image);
  waitKey(1);
  destroyWindow(name);
}

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;
  }
}
```

```cpp
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++)
  {
    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()
{
  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);

  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);

  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);

  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_object->OverlapsWith(current_recognised_object))
              {
                if (current_ground_truth_object->getName().compare(current_recognised_object->getName()) == 0)
                  overlapping_correct_objects.push_back(current_recognised_object);
                else
                  overlapping_incorrect_objects.push_back(current_recognised_object);
              }
            recognised_object_index++;
          }
          if ((overlapping_correct_objects.size() == 0) && (overlapping_incorrect_objects.size() == 0))
          {
            if (display_image != NULL)
            {
              Scalar colour(0x00, 0x00, 0xFF);
              current_ground_truth_object->DrawObject(display_image, colour);
            }
            results.AddFalseNegative(current_ground_truth_object->getName());
            cout << current_annotated_ground_truth_image->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, 0x00);
              results.AddMatch(current_ground_truth_object->getName(), overlapping_correct_objects[index]->getName(), (index > 0));
              if (index > 0)
              {
                colour[2] = 0xFF;
                cout << current_annotated_ground_truth_image->filename << ", " << current_ground_truth_object->getName() << ", (Duplicate) , " << current_ground_truth_object->getVerticesString() << endl;
              }
              if (display_image != NULL)
                current_ground_truth_object->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_objects[index]->DrawObject(display_image, colour);
              }
              results.AddMatch(current_ground_truth_object->getName(), overlapping_incorrect_objects[index]->getName(), (index > 0));
```

```
              cout << current_annotated_ground_truth_image->filename << ", " << current_ground_truth_object->getName() << ", (Mismatch), " << overlapping_incorrect_objects[index]->getName() << " , " << current_ground_truth_object->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_object);
            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, 0xFF);
              current_recognised_object->DrawObject(display_image, colour);
            }
            cout << current_annotated_recognition_image->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;
}


void ObjectAndLocation::setImage(Mat object_image)
{
  image = object_image.clone();
  // *** Student should add any initialisation (of their images or features; see private data below) they wish into this method.
}

```

```
Mat gaussianBlur(Mat image, string fileName)
{
  Mat blurredImg;
  GaussianBlur(image, blurredImg, Size(31, 31), 3, 3);
  //showImage("Gaussian blur " + fileName, blurredImg);
  return blurredImg;
}


Mat laplacian(Mat image, string fileName)
{
  // Create a kernel that we will use to sharpen our image
  Mat kernel = (Mat_<float>(3, 3) <<
    1, 1, 1,
    1, -8, 1,
    1, 1, 1);
  // an approximation of second derivative, a quite strong kernel
  // do the laplacian filtering as it is
  // well, we need to convert everything in something more deeper then CV_8U
  // because the kernel has some negative values,
  // and we can expect in general to have a Laplacian image with negative values
  // BUT a 8bits unsigned int (the one we are working with) can contain values
from 0 to 255
  // so the possible negative number will be truncated
  Mat imgLaplacian;
  filter2D(image, imgLaplacian, CV_32F, kernel);
  imgLaplacian.convertTo(imgLaplacian, CV_8UC3);
  showImage("Laplace Filtered Image " + fileName, imgLaplacian);
  return imgLaplacian;
}

Mat binary(Mat image, string fileName)
{
  Mat binaryImg;
  cvtColor(image, binaryImg, COLOR_BGR2GRAY);
  threshold(binaryImg, binaryImg, 40, 255, THRESH_BINARY | THRESH_OTSU);
  showImage("Binary Image " + fileName, binaryImg);
  return binaryImg;
}

Mat distanceTransform(Mat image, string fileName)
{
  // Perform the distance transform algorithm
  Mat distImg;
  distanceTransform(image, distImg, DIST_L2, 3);
  // Normalize the distance image for range = {0.0, 1.0}
  // so we can visualize and threshold it
  normalize(distImg, distImg, 0, 1.0, NORM_MINMAX);
  showImage("Distance Transform Image " + fileName, distImg);
  return distImg;
}

Mat waterShed(Mat image, Mat distImg, Mat dist_8u, string fileName)
{
  // Find total markers
  vector<vector<Point> > contours;
  findContours(dist_8u, contours, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE);
  // Create the marker image for the watershed algorithm
  Mat markers = Mat::zeros(distImg.size(), CV_32SC1);
  // Draw the foreground markers
  for (size_t i = 0; i < contours.size(); i++)
  {
```

```
    drawContours(markers, contours, static_cast<int>(i), Scalar(static_cast<int>(i) + 1), -1);
  }
  // Draw the background marker
  circle(markers, Point(5, 5), 3, Scalar(255), -1);
  //showImage("Markers " + fileName, markers * 10000);

  // Perform the watershed algorithm
  watershed(image, markers);
  Mat mark;
  markers.convertTo(mark, CV_8U);
  bitwise_not(mark, mark);
  //    imshow("Markers_v2", mark); // uncomment this if you want to see how the mark
  // image looks like at that point
  // Generate random colors
  vector<Vec3b> colors;
  for (size_t i = 0; i < contours.size(); i++)
  {
    int b = theRNG().uniform(0, 256);
    int g = theRNG().uniform(0, 256);
    int r = theRNG().uniform(0, 256);
    colors.push_back(Vec3b((uchar)b, (uchar)g, (uchar)r));
  }
  // Create the result image
  Mat segmentImg = Mat::zeros(markers.size(), CV_8UC3);
  // Fill labeled objects with random colors
  for (int i = 0; i < markers.rows; i++)
  {
    for (int j = 0; j < markers.cols; j++)
    {
      int index = markers.at<int>(i, j);
      if (index > 0 && index <= static_cast<int>(contours.size()))
      {
        segmentImg.at<Vec3b>(i, j) = colors[index - 1];
      }
    }
  }
  // Visualize the final image
  showImage("Watershed Segmented  " + fileName, segmentImg);

  return segmentImg;
}

Mat sharpen(Mat image, Mat laplacian, string fileName)
{
  Mat sharp;
  image.convertTo(sharp, CV_32F);
  laplacian.convertTo(laplacian, CV_32F);
  Mat sharpenedImg = sharp - laplacian;
  // convert back to 8bits gray scale
  sharpenedImg.convertTo(sharpenedImg, CV_8UC3);
  showImage("Sharpened Image " + fileName, sharpenedImg);
  return sharpenedImg;
}

Mat segmentRegions(Mat image, string fileName)
{
  Mat blurredImg = gaussianBlur(image, fileName);

  Mat lapacianImg = laplacian(image, fileName);

```

```
  //Mat sharpenedImg = sharpen(image, lapacianImg, fileName);

  Mat binaryImg = binary(lapacianImg, fileName);

  Mat distImg = distanceTransform(binaryImg, fileName);

  // Threshold to obtain the peaks
  // This will be the markers for the foreground objects
  threshold(distImg, distImg, 0.4, 1.0, THRESH_BINARY);

  // Dilate a bit the dist image
  Mat kernel1 = Mat::ones(3, 3, CV_8U);
  dilate(distImg, distImg, kernel1);
  showImage("Peaks " + fileName, distImg);

  // Create the CV_8U version of the distance image
  // It is needed for findContours()
  Mat dist_8u;
  distImg.convertTo(dist_8u, CV_8U);

  Mat segmentImg = waterShed(blurredImg, distImg, dist_8u, fileName);

  return segmentImg;
}

vector<RotatedRect> getBoxes(Mat image, string fileName)
{
  Size imgSize = image.size();

  // get contours
  vector<vector<Point>> contours;
  findContours(image, contours, RETR_TREE, CHAIN_APPROX_NONE);

  vector<vector<Point>> contours_poly;
  vector<RotatedRect> rotatedRect;

  size_t i = 0;
  while (true) {
    if (i >= contours.size()) {
      break;
    }
    vector<Point> tmp(contours[i].size());
    float epsilon = 0.1*arcLength(contours[i], true);
    approxPolyDP(contours[i], tmp, epsilon, true);
    if (tmp.size() == 4) {
      RotatedRect tmpRect = minAreaRect(tmp);

      if (tmpRect.size.height != 0 && tmpRect.size.width != 0) {
        if (0.66 < (tmpRect.size.height / tmpRect.size.width) < 1.5) {
          if (tmpRect.size.height * tmpRect.size.height > 0.001 * imgSize.height * imgSize.width) {
            contours_poly.push_back(tmp);
            rotatedRect.push_back(tmpRect);
          }
        }
      }
    }
    i++;
  }

  Mat drawing = Mat::zeros(image.size(), CV_8UC3);
  for (size_t i = 0; i < rotatedRect.size(); i++)
```

```
  {
    if (contours_poly[i].size() > 0) {
      Scalar color = Scalar(theRNG().uniform(0, 256), theRNG().uniform(0, 256), theRNG().uniform(0, 256));
      Point2f rect_points[4];
      rotatedRect[i].points(rect_points);
      for (int j = 0; j < 4; j++)
      {
        line(drawing, rect_points[j], rect_points[(j + 1) % 4], color);
      }
    }
  }

  showImage("Contours" + fileName, drawing);
  return rotatedRect;
}

bool containPoint(RotatedRect rectangle, Point2f point) {

  //Get the corner points.
  Point2f corners[4];
  rectangle.points(corners);

  //Convert the point array to a vector.
  Point2f* lastItemPointer = (corners + sizeof corners / sizeof corners[0]);
  vector<Point2f> contour(corners, lastItemPointer);

  //Check if the point is within the rectangle.
  double indicator = pointPolygonTest(contour, point, false);
  bool rectangleContainsPoint = (indicator > 0);
  return rectangleContainsPoint;
}

void ImageWithBlueSignObjects::LocateAndAddAllObjects(AnnotatedImages& training_images)
{
  cout << "Analysing" << filename << "...." << "\n";
  vector<ImageWithObjects *> train_objs = training_images.annotated_images;
  vector<pair<Mat, string>> train_objImages;
  for (int i = 0; i < train_objs.size(); i++) {
    ObjectAndLocation* obj = train_objs[i]->getObject(0);

    Mat img = obj->getImage();
    string className = obj->getName();
    resize(img, img, Size(200, 200));
    cvtColor(img, img, COLOR_BGR2GRAY);
    GaussianBlur(img, img, Size(5, 5), 0, 0);
    //adaptiveThreshold(img, img, 255, ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY_INV, 11, 2);
    threshold(img, img, 200, 255, THRESH_BINARY | THRESH_OTSU);

    getRectSubPix(img, Size(176, 176), Point2f(99.0, 99.0), img);

    Mat flipVertical;
    Mat flipHorizontal;
    Mat flipBoth;

    flip(img, flipVertical, 0);
    flip(img, flipHorizontal, 1);
    flip(img, flipBoth, -1);

    //showImage("train", img);
```

```
    //showImage("train", flipVertical);
    //showImage("train", flipHorizontal);
    //showImage("train", flipBoth);

    train_objImages.push_back(make_pair(img, className));
    train_objImages.push_back(make_pair(flipVertical, className));
    train_objImages.push_back(make_pair(flipHorizontal, className));
    train_objImages.push_back(make_pair(flipBoth, className));

    for (int angle = 90.0; angle < 360.0; angle += 90.0) {
      Mat M, rotated;
      M = getRotationMatrix2D(Point2f(87, 87), angle, 1.0);
      // perform the affine transformation
      warpAffine(img, rotated, M, img.size(), INTER_CUBIC);
      //showImage("train", rotated);
      train_objImages.push_back(make_pair(rotated, className));
    }

  }



  // Thresholding
  Mat gray;
  cvtColor(image, gray, COLOR_BGR2GRAY);
  Mat blurred = gaussianBlur(gray, filename);

  //threshold(blurred, blurred, 200, 255, THRESH_BINARY | THRESH_OTSU);
  adaptiveThreshold(blurred, blurred, 255, ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY_INV, 21, 2);
  //showImage("Threshold " + filename, blurred);



  //Mat out;
  //int thresh = 100;
  //Canny(blurred, out, thresh, thresh * 3);
  //showImage("Canny" + filename, out);

  vector<RotatedRect> possibleBoxes = getBoxes(blurred, filename);
  map < string, vector < pair<int, Mat> >> candidates;
  float match_thresh = 0.5;

  for (int i = 0; i < Classes.size(); i++) {
    candidates.insert({ Classes[i], vector<pair<int, Mat>>() });
  }

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


    Mat M, rotated, cropped;
    float angle = possibleBoxes[j].angle;
    Size rect_size = possibleBoxes[j].size;

    if (possibleBoxes[j].angle < -45.) {
      angle += 90.0;
      swap(rect_size.width, rect_size.height);
    }
    // get the rotation matrix
    M = getRotationMatrix2D(possibleBoxes[j].center, angle, 1.0);
    // perform the affine transformation
    warpAffine(image, rotated, M, image.size(), INTER_CUBIC);
```

```
    // crop the resulting image
    getRectSubPix(rotated, rect_size, possibleBoxes[j].center, cropped);

    Mat float_data, resized;
    resize(cropped, resized, Size(200, 200));
    cvtColor(resized, resized, COLOR_BGR2GRAY);
    GaussianBlur(resized, resized, Size(5, 5), 0, 0);
    //adaptiveThreshold(resized, resized, 255, ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY_INV, 11, 2);
    threshold(resized, resized, 200, 255, THRESH_BINARY | THRESH_OTSU);

    getRectSubPix(resized, Size(176, 176), Point2f(99.0, 99.0), resized);

    float val = 1;
    string classname;
    //if (filename == "Blue Signs/Testing/Blue020.jpg"){
    //  showImage(classname, resized);
    //}
    //
    for (int i = 0; i < train_objImages.size(); i++) {
      pair <Mat, string> curImg = train_objImages[i];
      Mat result;

      matchTemplate(resized, curImg.first, result, TM_SQDIFF_NORMED);
      auto r = result.at<float>(0);

      if (r < match_thresh && r < val) {
        val = r;
        classname = curImg.second;
      }
    }

    if (val < match_thresh) {
      candidates[classname].push_back(make_pair(j, resized));
      //showImage(classname, resized);
    }


  }

  vector<RotatedRect> boxes;

  for (auto it = candidates.begin(); it != candidates.end(); it++) {
    for (size_t i = 0; i < it->second.size(); i++) {

      RotatedRect box = possibleBoxes[it->second[i].first];
      boxes.push_back(box);
    }
  }

  for (auto it = candidates.begin(); it != candidates.end(); it++) {
    for (size_t i = 0; i < it->second.size(); i++) {

      RotatedRect box = possibleBoxes[it->second[i].first];
      cv::Point2f pts[4];
      box.points(pts);

      bool add = true;
      for (int j = 0; j < boxes.size(); j++) {
        if (containPoint(boxes[j], pts[0]) && containPoint(boxes[j], pts[1]) && containPoint(boxes[j], pts[2]) && containPoint(boxes[j], pts[3])) {
          add = false;
```

```
        }
      }
      if (add) {
        this->addObject(it->first, pts[0].x, pts[0].y, pts[1].x, pts[1].y, pts[2].x, pts[2].y, pts[3].x, pts[3].y, it->second[i].second);
      }
    }
  }

  //showImage(filename, image);
}


#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.
  return BAD_MATCHING_VALUE;
}
```