

```

1 #include "Utilities.h"
2 #include <iostream>
3 #include <fstream>
4 #include <list>
5 #include <experimental/filesystem> // C++-standard header file name
6 #include <filesystem> // Microsoft-specific implementation header file name
7 #include <opencv2/ml.hpp>
8 #include <map>
9 #include <string>
10
11 using namespace std::experimental::filesystem::v1;
12 using namespace std;
13
14
15 // Sign must be at least 100x100
16 #define MINIMUM_SIGN_SIDE 100
17 #define MINIMUM_SIGN_AREA 10000
18 #define MINIMUM_SIGN_BOUNDARY_LENGTH 400
19 #define STANDARD_SIGN_WIDTH_AND_HEIGHT 200
20 // Best match must be 10% better than second best match
21 #define REQUIRED_RATIO_OF_BEST_TO_SECOND_BEST 1.1
22 // Located shape must overlap the ground truth by 80% to be considered a match
23 #define REQUIRED_OVERLAP 0.8
24
25 vector<std::string> Classes = {
26     "Coffee",
27     "Disabled",
28     "Escalator",
29     "Exit",
30     "Gents",
31     "Information",
32     "Ladies",
33     "Lift",
34     "One",
35     "Stairs",
36     "TicketDesk",
37     "Two"
38 };
39
40 void showImage(string name, Mat image)
41 {
42     namedWindow(name, WINDOW_NORMAL);
43     imshow(name, image);
44     waitKey(1);
45     destroyWindow(name);
46 }
47
48 class ObjectAndLocation
49 {
50 public:
51     ObjectAndLocation(string object_name, Point top_left, Point top_right, Point
bottom_right, Point bottom_left, Mat object_image);
52     ObjectAndLocation(FileNode& node);
53     void write(FileStorage& fs);
54     void read(FileNode& node);
55     Mat& getImage();
56     string getName();
57     void setName(string new_name);
58     string getVerticesString();
59     void DrawObject(Mat* display_image, Scalar& colour);
60     double getMinimumSideLength();
61     double getArea();

```

```

62 void getVertice(int index, int& x, int& y);
63 void setImage(Mat image); // *** Student should add any initialisation (of
their images or features; see private data below) they wish into this method.
64 double compareObjects(ObjectAndLocation* otherObject); // *** Student should
write code to compare objects using chosen method.
65 bool OverlapsWith(ObjectAndLocation* other_object);
66 private:
67 string object_name;
68 Mat image;
69 vector<Point2i> vertices;
70 // *** Student can add whatever images or features they need to describe the
object.
71 };
72
73 class AnnotatedImages;
74
75 class ImageWithObjects
76 {
77     friend class AnnotatedImages;
78 public:
79     ImageWithObjects(string passed_filename);
80     ImageWithObjects(FileNode& node);
81     virtual void LocateAndAddAllObjects(AnnotatedImages& training_images) = 0;
82     ObjectAndLocation* addObject(string object_name, int top_left_column, int
top_left_row, int top_right_column, int top_right_row,
83     int bottom_right_column, int bottom_right_row, int bottom_left_column, int
bottom_left_row, Mat& image);
84     void write(FileStorage& fs);
85     void read(FileNode& node);
86     ObjectAndLocation* getObject(int index);
87     void extractAndSetObjectImage(ObjectAndLocation *new_object);
88     string ExtractObjectName(string filenamestr);
89     void FindBestMatch(ObjectAndLocation* new_object, string& object_name, double&
match_value);
90 protected:
91     string filename;
92     Mat image;
93     vector<ObjectAndLocation> objects;
94 };
95
96 class ImageWithBlueSignObjects : public ImageWithObjects
97 {
98 public:
99     ImageWithBlueSignObjects(string passed_filename);
100     ImageWithBlueSignObjects(FileNode& node);
101     void LocateAndAddAllObjects(AnnotatedImages& training_images); // *** Student
needs to develop this routine and add in objects using the addObject method
102 };
103
104 class ConfusionMatrix;
105
106 class AnnotatedImages
107 {
108 public:
109     AnnotatedImages(string directory_name);
110     AnnotatedImages();
111     void addAnnotatedImage(ImageWithObjects &annotated_image);
112     void write(FileStorage& fs);
113     void read(FileStorage& fs);
114     void read(FileNode& node);
115     void read(string filename);
116     void LocateAndAddAllObjects(AnnotatedImages& training_images);

```

```

117 void FindBestMatch(ObjectAndLocation* new_object);
118 Mat getImageOfAllObjects(int break_after = 7);
119 void CompareObjectsWithGroundTruth(AnnotatedImages& training_images,
AnnotatedImages& ground_truth, ConfusionMatrix& results);
120 ImageWithObjects* getAnnotatedImage(int index);
121 ImageWithObjects* FindAnnotatedImage(string filename_to_find);
122 public:
123     string name;
124     vector<ImageWithObjects*> annotated_images;
125 };
126
127 class ConfusionMatrix
128 {
129 public:
130     ConfusionMatrix(AnnotatedImages training_images);
131     void AddMatch(string ground_truth, string recognised_as, bool duplicate =
false);
132     void AddFalseNegative(string ground_truth);
133     void AddFalsePositive(string recognised_as);
134     void Print();
135 private:
136     void AddObjectClass(string object_class_name);
137     int getObjectClassIndex(string object_class_name);
138     vector<string> class_names;
139     int confusion_size;
140     int** confusion_matrix;
141     int false_index;
142     int tp, fp, fn;
143 };
144
145 ObjectAndLocation::ObjectAndLocation(string passed_object_name, Point top_left,
Point top_right, Point bottom_right, Point bottom_left, Mat object_image)
146 {
147     object_name = passed_object_name;
148     vertices.push_back(top_left);
149     vertices.push_back(top_right);
150     vertices.push_back(bottom_right);
151     vertices.push_back(bottom_left);
152     setImage(object_image);
153 }
154 ObjectAndLocation::ObjectAndLocation(FileNode& node)
155 {
156     read(node);
157 }
158 void ObjectAndLocation::write(FileStorage& fs)
159 {
160     fs << "{" << "nameStr" << object_name;
161     fs << "coordinates" << "[";
162     for (int i = 0; i < vertices.size(); ++i)
163     {
164         fs << "[:" << vertices[i].x << vertices[i].y << "];";
165     }
166     fs << "];";
167     fs << "}";
168 }
169 void ObjectAndLocation::read(FileNode& node)
170 {
171     node["nameStr"] >> object_name;
172     FileNode data = node["coordinates"];
173     for (FileNodeIterator itData = data.begin(); itData != data.end(); ++itData)
174     {
175         // Read each point

```

```

176     FileNode pt = *itData;
177
178     Point2i point;
179     FileNodeIterator itPt = pt.begin();
180     point.x = *itPt; ++itPt;
181     point.y = *itPt;
182     vertices.push_back(point);
183 }
184 }
185 Mat& ObjectAndLocation::getImage()
186 {
187     return image;
188 }
189 string ObjectAndLocation::getName()
190 {
191     return object_name;
192 }
193 void ObjectAndLocation::setName(string new_name)
194 {
195     object_name.assign(new_name);
196 }
197 string ObjectAndLocation::getVerticesString()
198 {
199     string result;
200     for (int index = 0; (index < vertices.size()); index++)
201         result.append("(" + to_string(vertices[index].x) + " " +
202             to_string(vertices[index].y) + ") ");
203     return result;
204 }
205 void ObjectAndLocation::DrawObject(Mat* display_image, Scalar& colour)
206 {
207     writeText(*display_image, object_name, vertices[0].y - 8, vertices[0].x + 8,
208         colour, 2.0, 4);
209     polylines(*display_image, vertices, true, colour, 8);
210 }
211 double ObjectAndLocation::getMinimumSideLength()
212 {
213     double min_distance = DistanceBetweenPoints(vertices[0],
214         vertices[vertices.size() - 1]);
215     for (int index = 0; (index < vertices.size() - 1); index++)
216     {
217         double distance = DistanceBetweenPoints(vertices[index], vertices[index + 1]);
218         if (distance < min_distance)
219             min_distance = distance;
220     }
221     return min_distance;
222 }
223 double ObjectAndLocation::getArea()
224 {
225     return contourArea(vertices);
226 }
227 void ObjectAndLocation::getVertice(int index, int& x, int& y)
228 {
229     if ((vertices.size() < index) || (index < 0))
230         x = y = -1;
231     else
232     {
233         x = vertices[index].x;
234         y = vertices[index].y;
235     }
236 }

```

```

235 ImageWithObjects::ImageWithObjects(string passed_filename)
236 {
237     filename = strdup(passed_filename.c_str());
238     cout << "Opening " << filename << endl;
239     image = imread(filename, -1);
240 }
241 ImageWithObjects::ImageWithObjects(FileNode& node)
242 {
243     read(node);
244 }
245 ObjectAndLocation* ImageWithObjects::addObject(string object_name, int
top_left_column, int top_left_row, int top_right_column, int top_right_row,
246     int bottom_right_column, int bottom_right_row, int bottom_left_column, int
bottom_left_row, Mat& image)
247 {
248     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);
249     objects.push_back(new_object);
250     return &(objects[objects.size() - 1]);
251 }
252 void ImageWithObjects::write(FileStorage& fs)
253 {
254     fs << "{" << "Filename" << filename << "Objects" << "[";
255     for (int index = 0; index < objects.size(); index++)
256         objects[index].write(fs);
257     fs << "]" << "}";
258 }
259 void ImageWithObjects::extractAndSetObjectImage(ObjectAndLocation *new_object)
260 {
261     Mat perspective_warped_image = Mat::zeros(STANDARD_SIGN_WIDTH_AND_HEIGHT,
STANDARD_SIGN_WIDTH_AND_HEIGHT, image.type());
262     Mat perspective_matrix(3, 3, CV_32FC1);
263     int x[4], y[4];
264     new_object->getVertice(0, x[0], y[0]);
265     new_object->getVertice(1, x[1], y[1]);
266     new_object->getVertice(2, x[2], y[2]);
267     new_object->getVertice(3, x[3], y[3]);
268     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]) }
};
269     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 } };
270     perspective_matrix = getPerspectiveTransform(source_points, destination_points);
271     warpPerspective(image, perspective_warped_image, perspective_matrix,
perspective_warped_image.size());
272     new_object->setImage(perspective_warped_image);
273 }
274 void ImageWithObjects::read(FileNode& node)
275 {
276     filename = (string)node["Filename"];
277     image = imread(filename, -1);
278     FileNode images_node = node["Objects"];
279     if (images_node.type() == FileNode::SEQ)
280     {
281         for (FileNodeIterator it = images_node.begin(); it != images_node.end(); ++it)
282         {
283             FileNode current_node = *it;
284             ObjectAndLocation *new_object = new ObjectAndLocation(current_node);
285             extractAndSetObjectImage(new_object);
286             objects.push_back(*new_object);

```

```

287     }
288 }
289 }
290 ObjectAndLocation* ImageWithObjects::getObject(int index)
291 {
292     if ((index < 0) || (index >= objects.size()))
293         return NULL;
294     else return &(objects[index]);
295 }
296 void ImageWithObjects::FindBestMatch(ObjectAndLocation* new_object, string&
object_name, double& match_value)
297 {
298     for (int index = 0; (index < objects.size()); index++)
299     {
300         double temp_match_score = objects[index].compareObjects(new_object);
301         if ((temp_match_score > 0.0) && ((match_value < 0.0) || (temp_match_score <
match_value)))
302         {
303             object_name = objects[index].getName();
304             match_value = temp_match_score;
305         }
306     }
307 }
308
309 string ImageWithObjects::ExtractObjectName(string filenamestr)
310 {
311     int last_slash = filenamestr.rfind("/");
312     int start_of_object_name = (last_slash == std::string::npos) ? 0 : last_slash +
1;
313     int extension = filenamestr.find(".", start_of_object_name);
314     int end_of_filename = (extension == std::string::npos) ? filenamestr.length() -
1 : extension - 1;
315     int end_of_object_name = filenamestr.find_last_not_of("1234567890",
end_of_filename);
316     end_of_object_name = (end_of_object_name == std::string::npos) ? end_of_filename
: end_of_object_name;
317     string object_name = filenamestr.substr(start_of_object_name, end_of_object_name
- start_of_object_name + 1);
318     return object_name;
319 }
320
321
322 ImageWithBlueSignObjects::ImageWithBlueSignObjects(string passed_filename) :
323     ImageWithObjects(passed_filename)
324 {
325 }
326 ImageWithBlueSignObjects::ImageWithBlueSignObjects(FileNode& node) :
327     ImageWithObjects(node)
328 {
329 }
330
331
332 AnnotatedImages::AnnotatedImages(string directory_name)
333 {
334     name = directory_name;
335     for (std::experimental::filesystem::directory_iterator
next(std::experimental::filesystem::path(directory_name.c_str())), end; next !=
end; ++next)
336     {
337         read(next->path().generic_string());
338     }
339 }

```

```

340 AnnotatedImages::AnnotatedImages()
341 {
342     name = "";
343 }
344 void AnnotatedImages::addAnnotatedImage(ImageWithObjects &annotated_image)
345 {
346     annotated_images.push_back(&annotated_image);
347 }
348
349 void AnnotatedImages::write(FileStorage& fs)
350 {
351     fs << "AnnotatedImages";
352     fs << "{";
353     fs << "name" << name << "ImagesAndObjects" << "[";
354     for (int index = 0; index < annotated_images.size(); index++)
355         annotated_images[index]->write(fs);
356     fs << "]" << "}";
357 }
358 void AnnotatedImages::read(FileStorage& fs)
359 {
360     FileNode node = fs.getFirstTopLevelNode();
361     read(node);
362 }
363 void AnnotatedImages::read(FileNode& node)
364 {
365     name = (string)node["name"];
366     FileNode images_node = node["ImagesAndObjects"];
367     if (images_node.type() == FileNode::SEQ)
368     {
369         for (FileNodeIterator it = images_node.begin(); it != images_node.end(); ++it)
370         {
371             FileNode current_node = *it;
372             ImageWithBlueSignObjects* new_image_with_objects = new
ImageWithBlueSignObjects(current_node);
373             annotated_images.push_back(new_image_with_objects);
374         }
375     }
376 }
377 void AnnotatedImages::read(string filename)
378 {
379     ImageWithBlueSignObjects *new_image_with_objects = new
ImageWithBlueSignObjects(filename);
380     annotated_images.push_back(new_image_with_objects);
381 }
382 void AnnotatedImages::LocateAndAddAllObjects(AnnotatedImages& training_images)
383 {
384     for (int index = 0; index < annotated_images.size(); index++)
385     {
386         annotated_images[index]->LocateAndAddAllObjects(training_images);
387     }
388 }
389 void AnnotatedImages::FindBestMatch(ObjectAndLocation* new_object) //Mat&
perspective_warped_image, string& object_name, double& match_value)
390 {
391     double match_value = -1.0;
392     string object_name = "Unknown";
393     double temp_best_match = 1000000.0;
394     string temp_best_name;
395     double temp_second_best_match = 1000000.0;
396     string temp_second_best_name;
397     for (int index = 0; index < annotated_images.size(); index++)
398     {

```

```

399     annotated_images[index]->FindBestMatch(new_object, object_name, match_value);
400     if (match_value < temp_best_match)
401     {
402         if (temp_best_name.compare(object_name) != 0)
403         {
404             temp_second_best_match = temp_best_match;
405             temp_second_best_name = temp_best_name;
406         }
407         temp_best_match = match_value;
408         temp_best_name = object_name;
409     }
410     else if ((match_value != temp_best_match) && (match_value <
temp_second_best_match) && (temp_best_name.compare(object_name) != 0))
411     {
412         temp_second_best_match = match_value;
413         temp_second_best_name = object_name;
414     }
415 }
416 if (temp_second_best_match / temp_best_match <
REQUIRED_RATIO_OF_BEST_TO_SECOND_BEST)
417     new_object->setName("Unknown");
418 else new_object->setName(temp_best_name);
419 }
420
421 Mat AnnotatedImages::getImageOfAllObjects(int break_after)
422 {
423     Mat all_rows_so_far;
424     Mat output;
425     int count = 0;
426     int object_index = 0;
427     string blank("");
428     for (int index = 0; (index < annotated_images.size()); index++)
429     {
430         ObjectAndLocation* current_object = NULL;
431         int object_index = 0;
432         while ((current_object = (annotated_images[index])->getObject(object_index))
!= NULL)
433         {
434             if (count == 0)
435             {
436                 output = JoinSingleImage(current_object->getImage(), current_object-
>getName());
437             }
438             else if (count % break_after == 0)
439             {
440                 if (count == break_after)
441                     all_rows_so_far = output;
442                 else
443                 {
444                     Mat temp_rows = JoinImagesVertically(all_rows_so_far, blank, output,
blank, 0);
445                     all_rows_so_far = temp_rows.clone();
446                 }
447                 output = JoinSingleImage(current_object->getImage(), current_object-
>getName());
448             }
449             else
450             {
451                 Mat new_output = JoinImagesHorizontally(output, blank, current_object-
>getImage(), current_object->getName(), 0);
452                 output = new_output.clone();
453             }

```



```

454         count++;
455         object_index++;
456     }
457 }
458 if (count == 0)
459 {
460     Mat blank_output(1, 1, CV_8UC3, Scalar(0, 0, 0));
461     return blank_output;
462 }
463 else if (count < break_after)
464     return output;
465 else {
466     Mat temp_rows = JoinImagesVertically(all_rows_so_far, blank, output, blank,
467     0);
468     all_rows_so_far = temp_rows.clone();
469     return all_rows_so_far;
470 }
471 }
472 ImageWithObjects* AnnotatedImages::getAnnotatedImage(int index)
473 {
474     if ((index >= 0) && (index < annotated_images.size()))
475         return annotated_images[index];
476     else return NULL;
477 }
478
479 ImageWithObjects* AnnotatedImages::FindAnnotatedImage(string filename_to_find)
480 {
481     for (int index = 0; (index < annotated_images.size()); index++)
482     {
483         if (filename_to_find.compare(annotated_images[index]->filename) == 0)
484             return annotated_images[index];
485     }
486     return NULL;
487 }
488
489 void MyApplication()
490 {
491     AnnotatedImages trainingImages;
492     FileStorage training_file("BlueSignsTraining.xml", FileStorage::READ);
493     if (!training_file.isOpened())
494     {
495         cout << "Could not open the file: \"" << "BlueSignsTraining.xml" << "\"" <<
endl;
496     }
497     else
498     {
499         trainingImages.read(training_file);
500     }
501     training_file.release();
502     //Mat image_of_all_training_objects = trainingImages.getImageOfAllObjects();
503     //imshow("All Training Objects", image_of_all_training_objects);
504     //imwrite("AllTrainingObjectImages.jpg", image_of_all_training_objects);
505     char ch = cv::waitKey(1);
506
507     AnnotatedImages groundTruthImages;
508     FileStorage ground_truth_file("BlueSignsGroundTruth.xml", FileStorage::READ);
509     if (!ground_truth_file.isOpened())
510     {
511         cout << "Could not open the file: \"" << "BlueSignsGroundTruth.xml" << "\"" <<
endl;
512     }

```

```

513     else
514     {
515         groundTruthImages.read(ground_truth_file);
516     }
517     ground_truth_file.release();
518     //Mat image_of_all_ground_truth_objects =
groundTruthImages.getImageOfAllObjects();
519     //imshow("All Ground Truth Objects", image_of_all_ground_truth_objects);
520     //imwrite("AllGroundTruthObjectImages.jpg", image_of_all_ground_truth_objects);
521     ch = cv::waitKey(1);
522
523     AnnotatedImages unknownImages("Blue Signs/Testing");
524     unknownImages.LocateAndAddAllObjects(trainingImages);
525     FileStorage unknowns_file("BlueSignsTesting.xml", FileStorage::WRITE);
526     if (!unknowns_file.isOpened())
527     {
528         cout << "Could not open the file: \"" << "BlueSignsTesting.xml" << "\"" <<
endl;
529     }
530     else
531     {
532         unknownImages.write(unknowns_file);
533     }
534     unknowns_file.release();
535     //Mat image_of_recognised_objects = unknownImages.getImageOfAllObjects();
536     //imshow("All Recognised Objects", image_of_recognised_objects);
537     //imwrite("AllRecognisedObjects.jpg", image_of_recognised_objects);
538
539     ConfusionMatrix results(trainingImages);
540     unknownImages.CompareObjectsWithGroundTruth(trainingImages, groundTruthImages,
results);
541     results.Print();
542 }
543
544
545 bool PointInPolygon(Point2i point, vector<Point2i> vertices)
546 {
547     int i, j, nvert = vertices.size();
548     bool inside = false;
549
550     for (i = 0, j = nvert - 1; i < nvert; j = i++)
551     {
552         if ((vertices[i].x == point.x) && (vertices[i].y == point.y))
553             return true;
554         if (((vertices[i].y >= point.y) != (vertices[j].y >= point.y)) &&
555             (point.x <= (vertices[j].x - vertices[i].x) * (point.y - vertices[i].y) /
(vertices[j].y - vertices[i].y) + vertices[i].x)
556             )
557             inside = !inside;
558     }
559     return inside;
560 }
561
562 bool ObjectAndLocation::OverlapsWith(ObjectAndLocation* other_object)
563 {
564     double area = contourArea(vertices);
565     double other_area = contourArea(other_object->vertices);
566     double overlap_area = 0.0;
567     int count_points_inside = 0;
568     for (int index = 0; (index < vertices.size()); index++)
569     {
570         if (PointInPolygon(vertices[index], other_object->vertices))

```

```

571     count_points_inside++;
572 }
573 int count_other_points_inside = 0;
574 for (int index = 0; (index < other_object->vertices.size()); index++)
575 {
576     if (PointInPolygon(other_object->vertices[index], vertices))
577         count_other_points_inside++;
578 }
579 if (count_points_inside == vertices.size())
580     overlap_area = area;
581 else if (count_other_points_inside == other_object->vertices.size())
582     overlap_area = other_area;
583 else if ((count_points_inside == 0) && (count_other_points_inside == 0))
584     overlap_area = 0.0;
585 else
586 { // There is a partial overlap of the polygons.
587     // Find min & max x & y for the current object
588     int min_x = vertices[0].x, min_y = vertices[0].y, max_x = vertices[0].x, max_y
= vertices[0].y;
589     for (int index = 0; (index < vertices.size()); index++)
590     {
591         if (min_x > vertices[index].x)
592             min_x = vertices[index].x;
593         else if (max_x < vertices[index].x)
594             max_x = vertices[index].x;
595         if (min_y > vertices[index].y)
596             min_y = vertices[index].y;
597         else if (max_y < vertices[index].y)
598             max_y = vertices[index].y;
599     }
600     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;
601     for (int index = 0; (index < other_object->vertices.size()); index++)
602     {
603         if (min_x2 > other_object->vertices[index].x)
604             min_x2 = other_object->vertices[index].x;
605         else if (max_x2 < other_object->vertices[index].x)
606             max_x2 = other_object->vertices[index].x;
607         if (min_y2 > other_object->vertices[index].y)
608             min_y2 = other_object->vertices[index].y;
609         else if (max_y2 < other_object->vertices[index].y)
610             max_y2 = other_object->vertices[index].y;
611     }
612     // We only need the maximum overlapping bounding boxes
613     if (min_x < min_x2) min_x = min_x2;
614     if (max_x > max_x2) max_x = max_x2;
615     if (min_y < min_y2) min_y = min_y2;
616     if (max_y > max_y2) max_y = max_y2;
617     // For all points
618     overlap_area = 0;
619     Point2i current_point;
620     // Try ever decreasing squares within the overlapping (image aligned) bounding
boxes to find the overlapping area.
621     bool all_points_inside = false;
622     int distance_from_edge = 0;
623     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++)
624     {
625         all_points_inside = true;
626         for (current_point.x = min_x + distance_from_edge; (current_point.x <=
(max_x - distance_from_edge)); current_point.x++)

```

```

627     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)
628     {
629         if ((PointInPolygon(current_point, vertices)) &&
(PointInPolygon(current_point, other_object->vertices)))
630             overlap_area++;
631         else all_points_inside = false;
632     }
633     for (current_point.y = min_y + distance_from_edge + 1; (current_point.y <=
(max_y - distance_from_edge - 1)); current_point.y++)
634     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)
635     {
636         if ((PointInPolygon(current_point, vertices)) &&
(PointInPolygon(current_point, other_object->vertices)))
637             overlap_area++;
638         else all_points_inside = false;
639     }
640 }
641 if (all_points_inside)
642     overlap_area += (max_x - min_x + 1 - 2 * (distance_from_edge + 1)) * (max_y
- min_y + 1 - 2 * (distance_from_edge + 1));
643 }
644 double percentage_overlap = (overlap_area*2.0) / (area + other_area);
645 return (percentage_overlap >= REQUIRED_OVERLAP);
646 }
647
648
649
650 void AnnotatedImages::CompareObjectsWithGroundTruth(AnnotatedImages&
training_images, AnnotatedImages& ground_truth, ConfusionMatrix& results)
651 {
652     // For every annotated image in ground_truth, find the corresponding image in
this
653     for (int ground_truth_image_index = 0; ground_truth_image_index <
ground_truth.annotated_images.size(); ground_truth_image_index++)
654     {
655         ImageWithObjects* current_annotated_ground_truth_image =
ground_truth.annotated_images[ground_truth_image_index];
656         ImageWithObjects* current_annotated_recognition_image =
FindAnnotatedImage(current_annotated_ground_truth_image->filename);
657         if (current_annotated_recognition_image != NULL)
658         {
659             ObjectAndLocation* current_ground_truth_object = NULL;
660             int ground_truth_object_index = 0;
661             Mat* display_image = NULL;
662             if (!current_annotated_recognition_image->image.empty())
663             {
664                 display_image = &(current_annotated_recognition_image->image);
665             }
666             // For each object in ground_truth.annotated_image
667             while ((current_ground_truth_object = current_annotated_ground_truth_image-
>getObject(ground_truth_object_index)) != NULL)
668             {
669                 if ((current_ground_truth_object->getMinimumSideLength() >=
MINIMUM_SIGN_SIDE) &&
670                     (current_ground_truth_object->getArea() >= MINIMUM_SIGN_AREA))
671                 {
672                     // Determine the number of overlapping objects (correct & incorrect)
673                     vector<ObjectAndLocation*> overlapping_correct_objects;

```

```

674         vector<ObjectAndLocation*> overlapping_incorrect_objects;
675         ObjectAndLocation* current_recognised_object = NULL;
676         int recognised_object_index = 0;
677         // For each object in this.annotated_image
678         while ((current_recognised_object = current_annotated_recognition_image-
>getObject(recognised_object_index)) != NULL)
679         {
680             if (current_recognised_object->getName().compare("Unknown") != 0)
681                 if (current_ground_truth_object-
>OverlapsWith(current_recognised_object))
682                 {
683                     if (current_ground_truth_object-
>getName().compare(current_recognised_object->getName()) == 0)
684                         overlapping_correct_objects.push_back(current_recognised_object);
685                     else
686                         overlapping_incorrect_objects.push_back(current_recognised_object);
687                     recognised_object_index++;
688                 }
689                 if ((overlapping_correct_objects.size() == 0) &&
(overlapping_incorrect_objects.size() == 0))
690                 {
691                     if (display_image != NULL)
692                     {
693                         Scalar colour(0x00, 0x00, 0xFF);
694                         current_ground_truth_object->DrawObject(display_image, colour);
695                     }
696                     results.AddFalseNegative(current_ground_truth_object->getName());
697                     cout << current_annotated_ground_truth_image->filename << ", " <<
current_ground_truth_object->getName() << ", (False Negative) , " <<
current_ground_truth_object->getVerticesString() << endl;
698                 }
699                 else {
700                     for (int index = 0; (index < overlapping_correct_objects.size());
index++)
701                     {
702                         Scalar colour(0x00, 0xFF, 0x00);
703                         results.AddMatch(current_ground_truth_object->getName(),
overlapping_correct_objects[index]->getName(), (index > 0));
704                         if (index > 0)
705                         {
706                             colour[2] = 0xFF;
707                             cout << current_annotated_ground_truth_image->filename << ", " <<
current_ground_truth_object->getName() << ", (Duplicate) , " <<
current_ground_truth_object->getVerticesString() << endl;
708                         }
709                         if (display_image != NULL)
710                             current_ground_truth_object->DrawObject(display_image, colour);
711                     }
712                     for (int index = 0; (index < overlapping_incorrect_objects.size());
index++)
713                     {
714                         if (display_image != NULL)
715                         {
716                             Scalar colour(0xFF, 0x00, 0xFF);
717                             overlapping_incorrect_objects[index]->DrawObject(display_image,
colour);
718                         }
719                         results.AddMatch(current_ground_truth_object->getName(),
overlapping_incorrect_objects[index]->getName(), (index > 0));

```

```

720         cout << current_annotated_ground_truth_image->filename << ", " <<
current_ground_truth_object->getName() << ", (Mismatch), " <<
overlapping_incorrect_objects[index]->getName() << ", " <<
current_ground_truth_object->getVerticesString() << endl;;
721     }
722 }
723 }
724 else
725     cout << current_annotated_ground_truth_image->filename << ", " <<
current_ground_truth_object->getName() << ", (DROPPED GT) , " <<
current_ground_truth_object->getVerticesString() << endl;
726
727     ground_truth_object_index++;
728 }
729 // For each object in this.annotated_image
730 //     For each overlapping object in ground_truth.annotated_image
731 //     Don't do anything (as already done above)
732 //     If no overlapping objects.
733 //     Update the confusion table (with a False Positive)
734 ObjectAndLocation* current_recognised_object = NULL;
735 int recognised_object_index = 0;
736 // For each object in this.annotated_image
737 while ((current_recognised_object = current_annotated_recognition_image-
>getObject(recognised_object_index)) != NULL)
738 {
739     if ((current_recognised_object->getMinimumSideLength() >=
MINIMUM_SIGN_SIDE) &&
740         (current_recognised_object->getArea() >= MINIMUM_SIGN_AREA))
741     {
742         // Determine the number of overlapping objects (correct & incorrect)
743         vector<ObjectAndLocation*> overlapping_objects;
744         ObjectAndLocation* current_ground_truth_object = NULL;
745         int ground_truth_object_index = 0;
746         // For each object in ground_truth.annotated_image
747         while ((current_ground_truth_object =
current_annotated_ground_truth_image->getObject(ground_truth_object_index)) !=
NULL)
748         {
749             if (current_ground_truth_object-
>OverlapsWith(current_recognised_object))
750                 overlapping_objects.push_back(current_ground_truth_object);
751             ground_truth_object_index++;
752         }
753         if ((overlapping_objects.size() == 0) && (current_recognised_object-
>getName().compare("Unknown") != 0))
754         {
755             results.AddFalsePositive(current_recognised_object->getName());
756             if (display_image != NULL)
757             {
758                 Scalar colour(0x7F, 0x7F, 0xFF);
759                 current_recognised_object->DrawObject(display_image, colour);
760             }
761             cout << current_annotated_recognition_image->filename << ", " <<
current_recognised_object->getName() << ", (False Positive) , " <<
current_recognised_object->getVerticesString() << endl;
762         }
763     }
764     else
765     {
766         cout << current_annotated_recognition_image->filename << ", " <<
current_recognised_object->getName() << ", (DROPPED) , " <<
current_recognised_object->getVerticesString() << endl;
766         recognised_object_index++;

```

```

767     }
768     if (display_image != NULL)
769     {
770         Mat smaller_image;
771         resize(*display_image, smaller_image, Size(display_image->cols / 4,
display_image->rows / 4));
772         imshow(current_annotated_recognition_image->filename, smaller_image);
773         char ch = cv::waitKey(1);
774         //         delete display_image;
775     }
776 }
777 }
778 }
779
780 // Determine object classes from the training_images (vector of strings)
781 // Create and zero a confusion matrix
782 ConfusionMatrix::ConfusionMatrix(AnnotatedImages training_images)
783 {
784     // Extract object class names
785     ImageWithObjects* current_annotated_image = NULL;
786     int image_index = 0;
787     while ((current_annotated_image =
training_images.getAnnotatedImage(image_index)) != NULL)
788     {
789         ObjectAndLocation* current_object = NULL;
790         int object_index = 0;
791         while ((current_object = current_annotated_image->getObject(object_index)) !=
NULL)
792         {
793             AddObjectClass(current_object->getName());
794             object_index++;
795         }
796         image_index++;
797     }
798     // Create and initialise confusion matrix
799     confusion_size = class_names.size() + 1;
800     confusion_matrix = new int*[confusion_size];
801     for (int index = 0; (index < confusion_size); index++)
802     {
803         confusion_matrix[index] = new int[confusion_size];
804         for (int index2 = 0; (index2 < confusion_size); index2++)
805             confusion_matrix[index][index2] = 0;
806     }
807     false_index = confusion_size - 1;
808 }
809 void ConfusionMatrix::AddObjectClass(string object_class_name)
810 {
811     int index = getObjectClassIndex(object_class_name);
812     if (index == -1)
813         class_names.push_back(object_class_name);
814     tp = fp = fn = 0;
815 }
816 int ConfusionMatrix::getObjectClassIndex(string object_class_name)
817 {
818     int index = 0;
819     for (; (index < class_names.size()) &&
(object_class_name.compare(class_names[index]) != 0); index++)
820     ;
821     if (index < class_names.size())
822         return index;
823     else return -1;
824 }

```



```

825 void ConfusionMatrix::AddMatch(string ground_truth, string recognised_as, bool
duplicate)
826 {
827     if ((ground_truth.compare(recognised_as) == 0) && (duplicate))
828         AddFalsePositive(recognised_as);
829     else
830     {
831         confusion_matrix[getObjectClassIndex(ground_truth)]
[getObjectClassIndex(recognised_as)]++;
832         if (ground_truth.compare(recognised_as) == 0)
833             tp++;
834         else {
835             fp++;
836             fn++;
837         }
838     }
839 }
840 void ConfusionMatrix::AddFalseNegative(string ground_truth)
841 {
842     fn++;
843     confusion_matrix[getObjectClassIndex(ground_truth)][false_index]++;
844 }
845 void ConfusionMatrix::AddFalsePositive(string recognised_as)
846 {
847     fp++;
848     confusion_matrix[false_index][getObjectClassIndex(recognised_as)]++;
849 }
850 void ConfusionMatrix::Print()
851 {
852     cout << ".,.,Recognised as:" << endl << ".,,";
853     for (int recognised_as_index = 0; recognised_as_index < confusion_size;
recognised_as_index++)
854         if (recognised_as_index < confusion_size - 1)
855             cout << class_names[recognised_as_index] << ",";
856         else cout << "False Negative,";
857     cout << endl;
858     for (int ground_truth_index = 0; (ground_truth_index <= class_names.size());
ground_truth_index++)
859     {
860         if (ground_truth_index < confusion_size - 1)
861             cout << "Ground Truth," << class_names[ground_truth_index] << ",";
862         else cout << "Ground Truth,False Positive,";
863         for (int recognised_as_index = 0; recognised_as_index < confusion_size;
recognised_as_index++)
864             cout << confusion_matrix[ground_truth_index][recognised_as_index] << ",";
865         cout << endl;
866     }
867     double precision = ((double)tp) / ((double)(tp + fp));
868     double recall = ((double)tp) / ((double)(tp + fn));
869     double f1 = 2.0*precision*recall / (precision + recall);
870     cout << endl << "Precision = " << precision << endl << "Recall = " << recall <<
endl << "F1 = " << f1 << endl;
871 }
872
873
874 void ObjectAndLocation::setImage(Mat object_image)
875 {
876     image = object_image.clone();
877     // *** Student should add any initialisation (of their images or features; see
private data below) they wish into this method.
878 }
879

```



```

880 Mat gaussianBlur(Mat image, string fileName)
881 {
882     Mat blurredImg;
883     GaussianBlur(image, blurredImg, Size(31, 31), 3, 3);
884     //showImage("Gaussian blur " + fileName, blurredImg);
885     return blurredImg;
886 }
887
888
889 Mat laplacian(Mat image, string fileName)
890 {
891     // Create a kernel that we will use to sharpen our image
892     Mat kernel = (Mat_<float>(3, 3) <<
893         1, 1, 1,
894         1, -8, 1,
895         1, 1, 1);
896     // an approximation of second derivative, a quite strong kernel
897     // do the laplacian filtering as it is
898     // well, we need to convert everything in something more deeper then CV_8U
899     // because the kernel has some negative values,
900     // and we can expect in general to have a Laplacian image with negative values
901     // BUT a 8bits unsigned int (the one we are working with) can contain values
    from 0 to 255
902     // so the possible negative number will be truncated
903     Mat imgLaplacian;
904     filter2D(image, imgLaplacian, CV_32F, kernel);
905     imgLaplacian.convertTo(imgLaplacian, CV_8UC3);
906     showImage("Laplace Filtered Image " + fileName, imgLaplacian);
907     return imgLaplacian;
908 }
909
910 Mat binary(Mat image, string fileName)
911 {
912     Mat binaryImg;
913     cvtColor(image, binaryImg, COLOR_BGR2GRAY);
914     threshold(binaryImg, binaryImg, 40, 255, THRESH_BINARY | THRESH_OTSU);
915     showImage("Binary Image " + fileName, binaryImg);
916     return binaryImg;
917 }
918
919 Mat distanceTransform(Mat image, string fileName)
920 {
921     // Perform the distance transform algorithm
922     Mat distImg;
923     distanceTransform(image, distImg, DIST_L2, 3);
924     // Normalize the distance image for range = {0.0, 1.0}
925     // so we can visualize and threshold it
926     normalize(distImg, distImg, 0, 1.0, NORM_MINMAX);
927     showImage("Distance Transform Image " + fileName, distImg);
928     return distImg;
929 }
930
931 Mat waterShed(Mat image, Mat distImg, Mat dist_8u, string fileName)
932 {
933     // Find total markers
934     vector<vector<Point> > contours;
935     findContours(dist_8u, contours, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE);
936     // Create the marker image for the watershed algorithm
937     Mat markers = Mat::zeros(distImg.size(), CV_32SC1);
938     // Draw the foreground markers
939     for (size_t i = 0; i < contours.size(); i++)
940     {

```

```

941     drawContours(markers, contours, static_cast<int>(i), Scalar(static_cast<int>
(i) + 1), -1);
942 }
943 // Draw the background marker
944 circle(markers, Point(5, 5), 3, Scalar(255), -1);
945 //showImage("Markers " + fileName, markers * 10000);
946
947 // Perform the watershed algorithm
948 watershed(image, markers);
949 Mat mark;
950 markers.convertTo(mark, CV_8U);
951 bitwise_not(mark, mark);
952 //    imshow("Markers_v2", mark); // uncomment this if you want to see how the
mark
953 // image looks like at that point
954 // Generate random colors
955 vector<Vec3b> colors;
956 for (size_t i = 0; i < contours.size(); i++)
957 {
958     int b = theRNG().uniform(0, 256);
959     int g = theRNG().uniform(0, 256);
960     int r = theRNG().uniform(0, 256);
961     colors.push_back(Vec3b((uchar)b, (uchar)g, (uchar)r));
962 }
963 // Create the result image
964 Mat segmentImg = Mat::zeros(markers.size(), CV_8UC3);
965 // Fill labeled objects with random colors
966 for (int i = 0; i < markers.rows; i++)
967 {
968     for (int j = 0; j < markers.cols; j++)
969     {
970         int index = markers.at<int>(i, j);
971         if (index > 0 && index <= static_cast<int>(contours.size()))
972         {
973             segmentImg.at<Vec3b>(i, j) = colors[index - 1];
974         }
975     }
976 }
977 // Visualize the final image
978 showImage("Watershed Segmented " + fileName, segmentImg);
979
980 return segmentImg;
981 }
982
983 Mat sharpen(Mat image, Mat laplacian, string fileName)
984 {
985     Mat sharp;
986     image.convertTo(sharp, CV_32F);
987     laplacian.convertTo(laplacian, CV_32F);
988     Mat sharpenedImg = sharp - laplacian;
989     // convert back to 8bits gray scale
990     sharpenedImg.convertTo(sharpenedImg, CV_8UC3);
991     showImage("Sharpened Image " + fileName, sharpenedImg);
992     return sharpenedImg;
993 }
994
995 Mat segmentRegions(Mat image, string fileName)
996 {
997     Mat blurredImg = gaussianBlur(image, fileName);
998
999     Mat laplacianImg = laplacian(image, fileName);
1000

```

```

1001 //Mat sharpenedImg = sharpen(image, lapacianImg, fileName);
1002
1003 Mat binaryImg = binary(lapacianImg, fileName);
1004
1005 Mat distImg = distanceTransform(binaryImg, fileName);
1006
1007 // Threshold to obtain the peaks
1008 // This will be the markers for the foreground objects
1009 threshold(distImg, distImg, 0.4, 1.0, THRESH_BINARY);
1010
1011 // Dilate a bit the dist image
1012 Mat kernel1 = Mat::ones(3, 3, CV_8U);
1013 dilate(distImg, distImg, kernel1);
1014 showImage("Peaks " + fileName, distImg);
1015
1016 // Create the CV_8U version of the distance image
1017 // It is needed for findContours()
1018 Mat dist_8u;
1019 distImg.convertTo(dist_8u, CV_8U);
1020
1021 Mat segmentImg = watershed(blurredImg, distImg, dist_8u, fileName);
1022
1023 return segmentImg;
1024 }
1025
1026 vector<RotatedRect> getBoxes(Mat image, string fileName)
1027 {
1028     Size imgSize = image.size();
1029
1030     // get contours
1031     vector<vector<Point>> contours;
1032     findContours(image, contours, RETR_TREE, CHAIN_APPROX_NONE);
1033
1034     vector<vector<Point>> contours_poly;
1035     vector<RotatedRect> rotatedRect;
1036
1037     size_t i = 0;
1038     while (true) {
1039         if (i >= contours.size()) {
1040             break;
1041         }
1042         vector<Point> tmp(contours[i].size());
1043         float epsilon = 0.1*arcLength(contours[i], true);
1044         approxPolyDP(contours[i], tmp, epsilon, true);
1045         if (tmp.size() == 4) {
1046             RotatedRect tmpRect = minAreaRect(tmp);
1047
1048             if (tmpRect.size.height != 0 && tmpRect.size.width != 0) {
1049                 if (0.66 < (tmpRect.size.height / tmpRect.size.width) < 1.5) {
1050                     if (tmpRect.size.height * tmpRect.size.height > 0.001 * imgSize.height *
imgSize.width) {
1051                         contours_poly.push_back(tmp);
1052                         rotatedRect.push_back(tmpRect);
1053                     }
1054                 }
1055             }
1056         }
1057         i++;
1058     }
1059
1060     Mat drawing = Mat::zeros(image.size(), CV_8UC3);
1061     for (size_t i = 0; i < rotatedRect.size(); i++)

```

```

1062 {
1063     if (contours_poly[i].size() > 0) {
1064         Scalar color = Scalar(theRNG().uniform(0, 256), theRNG().uniform(0, 256),
theRNG().uniform(0, 256));
1065         Point2f rect_points[4];
1066         rotatedRect[i].points(rect_points);
1067         for (int j = 0; j < 4; j++)
1068             {
1069                 line(drawing, rect_points[j], rect_points[(j + 1) % 4], color);
1070             }
1071     }
1072 }
1073
1074 showImage("Contours" + fileName, drawing);
1075 return rotatedRect;
1076 }
1077
1078 bool containPoint(RotatedRect rectangle, Point2f point) {
1079
1080     //Get the corner points.
1081     Point2f corners[4];
1082     rectangle.points(corners);
1083
1084     //Convert the point array to a vector.
1085     Point2f* lastItemPointer = (corners + sizeof corners / sizeof corners[0]);
1086     vector<Point2f> contour(corners, lastItemPointer);
1087
1088     //Check if the point is within the rectangle.
1089     double indicator = pointPolygonTest(contour, point, false);
1090     bool rectangleContainsPoint = (indicator > 0);
1091     return rectangleContainsPoint;
1092 }
1093
1094 void ImageWithBlueSignObjects::LocateAndAddAllObjects(AnnotatedImages&
training_images)
1095 {
1096     cout << "Analysing" << filename << "...." << "\n";
1097     vector<ImageWithObjects*> train_objs = training_images.annotated_images;
1098     vector<pair<Mat, string>> train_objImages;
1099     for (int i = 0; i < train_objs.size(); i++) {
1100         ObjectAndLocation* obj = train_objs[i]->getObject(0);
1101
1102         Mat img = obj->getImage();
1103         string className = obj->getName();
1104         resize(img, img, Size(200, 200));
1105         cvtColor(img, img, COLOR_BGR2GRAY);
1106         GaussianBlur(img, img, Size(5, 5), 0, 0);
1107         //adaptiveThreshold(img, img, 255, ADAPTIVE_THRESH_GAUSSIAN_C,
THRESH_BINARY_INV, 11, 2);
1108         threshold(img, img, 200, 255, THRESH_BINARY | THRESH_OTSU);
1109
1110         getRectSubPix(img, Size(176, 176), Point2f(99.0, 99.0), img);
1111
1112         Mat flipVertical;
1113         Mat flipHorizontal;
1114         Mat flipBoth;
1115
1116         flip(img, flipVertical, 0);
1117         flip(img, flipHorizontal, 1);
1118         flip(img, flipBoth, -1);
1119
1120         //showImage("train", img);

```

```

1121 //showImage("train", flipVertical);
1122 //showImage("train", flipHorizontal);
1123 //showImage("train", flipBoth);
1124
1125 train_objImages.push_back(make_pair(img, className));
1126 train_objImages.push_back(make_pair(flipVertical, className));
1127 train_objImages.push_back(make_pair(flipHorizontal, className));
1128 train_objImages.push_back(make_pair(flipBoth, className));
1129
1130 for (int angle = 90.0; angle < 360.0; angle += 90.0) {
1131     Mat M, rotated;
1132     M = getRotationMatrix2D(Point2f(87, 87), angle, 1.0);
1133     // perform the affine transformation
1134     warpAffine(img, rotated, M, img.size(), INTER_CUBIC);
1135     //showImage("train", rotated);
1136     train_objImages.push_back(make_pair(rotated, className));
1137 }
1138
1139 }
1140
1141
1142
1143 // Thresholding
1144 Mat gray;
1145 cvtColor(image, gray, COLOR_BGR2GRAY);
1146 Mat blurred = gaussianBlur(gray, filename);
1147
1148 //threshold(blurred, blurred, 200, 255, THRESH_BINARY | THRESH_OTSU);
1149 adaptiveThreshold(blurred, blurred, 255, ADAPTIVE_THRESH_GAUSSIAN_C,
THRESH_BINARY_INV, 21, 2);
1150 //showImage("Threshold " + filename, blurred);
1151
1152
1153
1154 //Mat out;
1155 //int thresh = 100;
1156 //Canny(blurred, out, thresh, thresh * 3);
1157 //showImage("Canny" + filename, out);
1158
1159 vector<RotatedRect> possibleBoxes = getBoxes(blurred, filename);
1160 map < string, vector < pair<int, Mat> >> candidates;
1161 float match_thresh = 0.5;
1162
1163 for (int i = 0; i < Classes.size(); i++) {
1164     candidates.insert({ Classes[i], vector<pair<int, Mat>>() });
1165 }
1166
1167 for (int j = 0; j < possibleBoxes.size(); j++) {
1168
1169
1170     Mat M, rotated, cropped;
1171     float angle = possibleBoxes[j].angle;
1172     Size rect_size = possibleBoxes[j].size;
1173
1174     if (possibleBoxes[j].angle < -45.) {
1175         angle += 90.0;
1176         swap(rect_size.width, rect_size.height);
1177     }
1178     // get the rotation matrix
1179     M = getRotationMatrix2D(possibleBoxes[j].center, angle, 1.0);
1180     // perform the affine transformation
1181     warpAffine(image, rotated, M, image.size(), INTER_CUBIC);

```

```

1182 // crop the resulting image
1183 getRectSubPix(rotated, rect_size, possibleBoxes[j].center, cropped);
1184
1185 Mat float_data, resized;
1186 resize(cropped, resized, Size(200, 200));
1187 cvtColor(resized, resized, COLOR_BGR2GRAY);
1188 GaussianBlur(resized, resized, Size(5, 5), 0, 0);
1189 //adaptiveThreshold(resized, resized, 255, ADAPTIVE_THRESH_GAUSSIAN_C,
THRESH_BINARY_INV, 11, 2);
1190 threshold(resized, resized, 200, 255, THRESH_BINARY | THRESH_OTSU);
1191
1192 getRectSubPix(resized, Size(176, 176), Point2f(99.0, 99.0), resized);
1193
1194 float val = 1;
1195 string classname;
1196 //if (filename == "Blue Signs/Testing/Blue020.jpg"){
1197 // showImage(classname, resized);
1198 //}
1199 //
1200 for (int i = 0; i < train_objImages.size(); i++) {
1201     pair <Mat, string> curImg = train_objImages[i];
1202     Mat result;
1203
1204     matchTemplate(resized, curImg.first, result, TM_SQDIFF_NORMED);
1205     auto r = result.at<float>(0);
1206
1207     if (r < match_thresh && r < val) {
1208         val = r;
1209         classname = curImg.second;
1210     }
1211 }
1212
1213 if (val < match_thresh) {
1214     candidates[classname].push_back(make_pair(j, resized));
1215     //showImage(classname, resized);
1216 }
1217
1218
1219 }
1220
1221 vector<RotatedRect> boxes;
1222
1223 for (auto it = candidates.begin(); it != candidates.end(); it++) {
1224     for (size_t i = 0; i < it->second.size(); i++) {
1225
1226         RotatedRect box = possibleBoxes[it->second[i].first];
1227         boxes.push_back(box);
1228     }
1229 }
1230
1231 for (auto it = candidates.begin(); it != candidates.end(); it++) {
1232     for (size_t i = 0; i < it->second.size(); i++) {
1233
1234         RotatedRect box = possibleBoxes[it->second[i].first];
1235         cv::Point2f pts[4];
1236         box.points(pts);
1237
1238         bool add = true;
1239         for (int j = 0; j < boxes.size(); j++) {
1240             if (containPoint(boxes[j], pts[0]) && containPoint(boxes[j], pts[1]) &&
containPoint(boxes[j], pts[2]) && containPoint(boxes[j], pts[3])) {
1241                 add = false;

```

```
1242     }
1243 }
1244 if (add) {
1245     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);
1246 }
1247 }
1248 }
1249
1250 //showImage(filename, image);
1251 }
1252
1253
1254 #define BAD_MATCHING_VALUE 1000000000.0;
1255 double ObjectAndLocation::compareObjects(ObjectAndLocation* otherObject)
1256 {
1257     // *** Student should write code to compare objects using chosen method.
1258     // Please bear in mind that ImageWithObjects::FindBestMatch assumes that the
lower the value the better. Feel free to change this.
1259     return BAD_MATCHING_VALUE;
1260 }
```