TSB-funded Project 'TADD' - Trainable vision-based anomaly detection and diagnosis Technical Report for September 2014

Ran Song and Tom Duckett

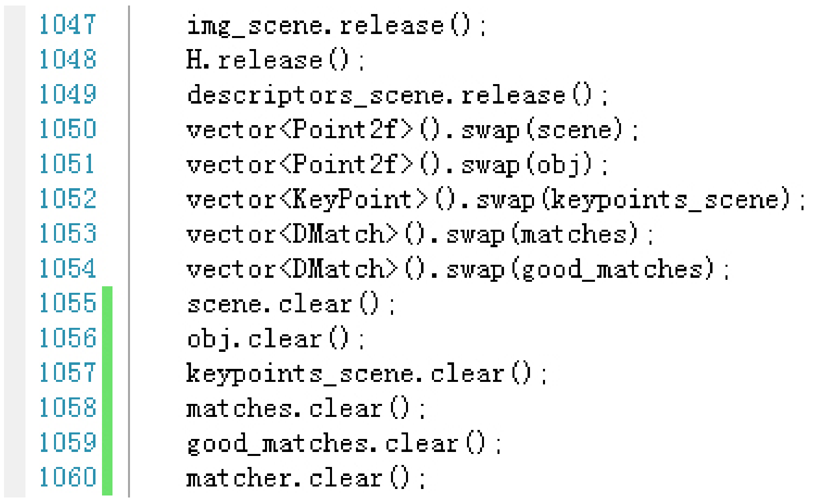
Agri-Food Technology Research Group, Lincoln Centre for Autonomous Systems Research, School of Computer Science, University of Lincoln, UK

Abstract

In September 2014, I worked for both Label-TADD and QA-TADD. Thus this report is composed of two parts. The work related to the Label-TADD includes software debugging and efficiency test. The work about the QA-TADD is the computation of the confusion matrix used for evaluating the performance in the tasks of multi-label classification.

1. Label-TADD debugging and updating

Occasionally, we found a bug within the original Label-TADD software. According to its specification, once the training has been done, it should be able to handle an indefinite number of test images where label region is detected in each test and OCR is also implemented. However, we found that after processing 6–8 images, the software would suffer from a crash. We manage to pinpoint the bug. It is caused by an accumulated change added to two vital parameters used as thresholds which control the number of corresponding SURF feature points (matching points). After 6–8 runs, these threshold become very intolerant and tend to reject all of the candidate matches. Consequently, there is no input data available for a built-in function of OpenCV which computes the homography (the transform matrix describing the motion between two images) based on the matching points. However, in terms of computer vision textbook, the computation of the homography requires at least four pairs of noncolinear matching points. As a result, when we call this function in our codes, the software crashes and no reason is given. This



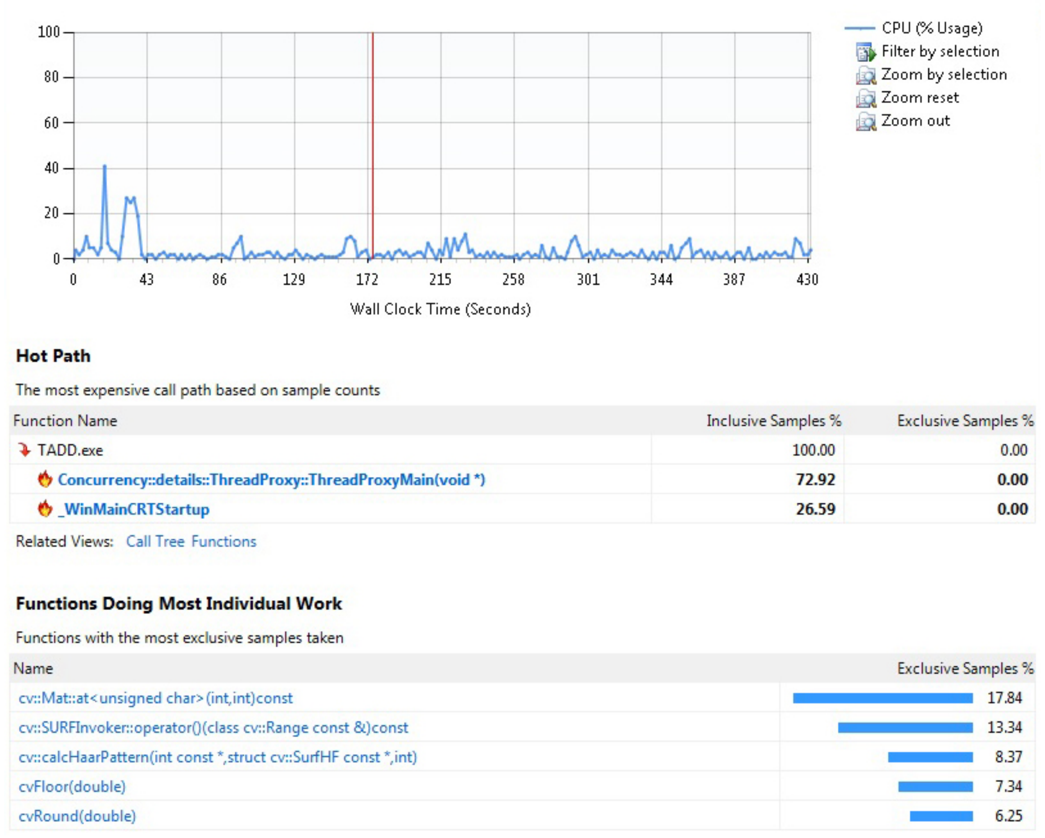
```
img_scene.release();
H.release();
descriptors_scene.release();
vector<Point2f>().swap(scene);
vector<Point2f>().swap(obj);
vector<KeyPoint>().swap(keypoints_scene);
vector<DMatch>().swap(matches);
vector<DMatch>().swap(good_matches);
scene.clear();
obj.clear();
keypoints_scene.clear();
matches.clear();
good_matches.clear();
matcher.clear();
```

Figure 1: A piece of codes for solving memory leak

is usual for open source libraries such OpenCV where the input check for its built-in functions sometimes is missing. Once we pinpoint the bug and find out how it is generated, the solution is simple. In the updated codes, we always reset these two parameters to default values at the beginning of every run.

Also, we found that the original software would slow down a little bit (by 30% roughly) after processing a large number of images. We also managed to sort this problem. It is mainly caused by memory leak, which makes the accessible memory decreased. In other words, although my desktop computer has 16GB installed memory (RAM), after keeping running for a while, it is equivalent that the software is being implemented on a computer with 11GB installed memory. Fig. 1 shows a piece of the newly added codes for avoiding memory leak. Fig. 2 shows the efficiency analysis of the updated version of the software. The running time of a specific function is proportional to the percentage of exclusive samples. It can be seen that there has been little room for the current program to further accelerate since the most time-consuming functions are OpenCV built-in functions which we assume have been optimised by its supporting team.

We have also done some initial analysis for the performance of the current software which now typically spends less than a second to process one test image with a resolution of 1197×777. We also suggest that after closing the GUI for exiting the software, it is better to make a double check through task



Hot Path

The most expensive call path based on sample counts

Function Name	Inclusive Samples %	Exclusive Samples %
TADD.exe	100.00	0.00
Concurrency::details::ThreadProxy::ThreadProxyMain(void *)	**72.92**	**0.00**
_WinMainCRTStartup	**26.59**	**0.00**

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Functions with the most exclusive samples taken

Name	Exclusive Samples %
cv::Mat::at<unsigned char>(int,int)const	17.84
cv::SURFInvoker::operator()(class cv::Range const &)const	13.34
cv::calcHaarPattern(int const *,struct cv::SurfHF const *,int)	8.37
cvFloor(double)	7.34
cvRound(double)	6.25

Figure 2: Efficiency analysis of the program where the running time of a function is proportional to its corresponding exclusive samples.

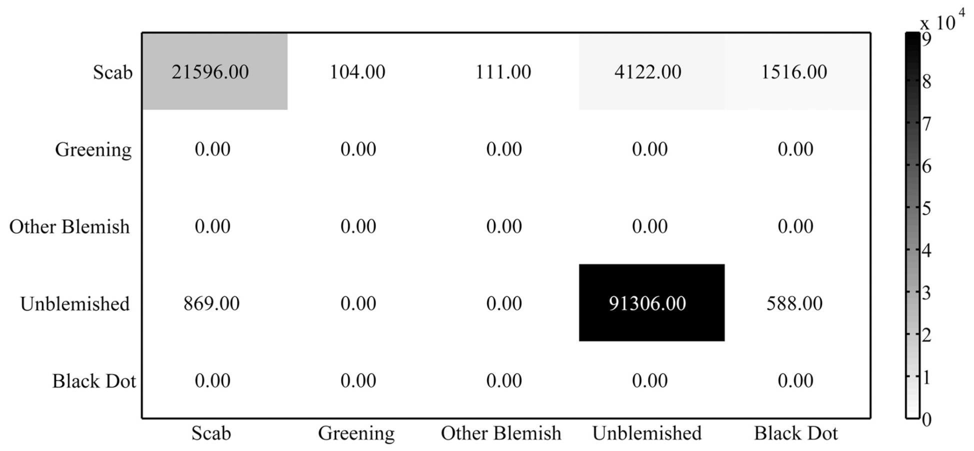
manager and terminate 'TADD.EXE' in the 'Processes' list to completely release the memory if it exists.

2. QA-TADD Evaluation

In September, we also implemented quantitative evaluation for the QA-TADD system. We computed the confusion matrices in order to evaluate the performance of the QA-TADD in the tasks of multi-label classification. In the field of machine learning, a confusion matrix, also known as a contingency table or an error matrix, is a specific table layout that allows visualisation of the performance of an algorithm, typically a supervised learning one. Each column of the matrix represents the instances in a predicted class, while each row represents the instances in an actual class. The name stems from the fact that it makes it easy to see if the system is confusing two classes. Confusion matrix is also a widely used statistic in computer vision to evaluate the accuracy of the classification. More importantly, it tells us what really happens when something goes wrong, which usually helps to make potential improvement aiming at the shortcomings of the current method.

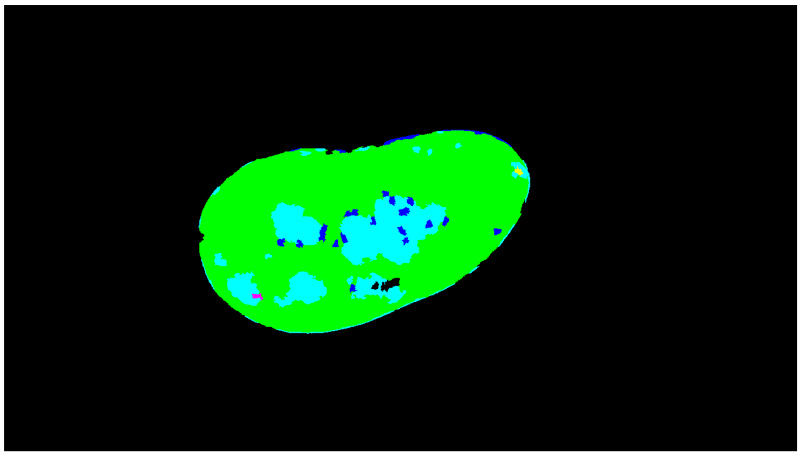
We computed confusion matrices for a set of 41 images which contain different types of potato defects such as scab, greening, black dot and other types of blemishes. Figs. 3–8 show the resultant confusion matrices of 6 input images. For example, to understand confusion matrix, in Fig. 3, 21596 pixels detected as 'Scab' are truly 'Scab' pixels while 869 pixels detected as 'scab' are actually 'Unblemished' pixels. Also in Fig. 3, there should be no 'Greening' pixels but 104 'Scab' pixels are incorrectly classified as 'Greening' pixels. This is consistent with the classification results and the ground truth shown in Fig. 3.

It can be seen that typically, there are quite a lot of pixels correctly labelled as 'Unblemished'. This suggests that the system should have an excellent performance in the tasks of two-label classification.



	Scab	Greening	Other Blemish	Unblemished	Black Dot
Scab	21596.00	104.00	111.00	4122.00	1516.00
Greening	0.00	0.00	0.00	0.00	0.00
Other Blemish	0.00	0.00	0.00	0.00	0.00
Unblemished	869.00	0.00	0.00	91306.00	588.00
Black Dot	0.00	0.00	0.00	0.00	0.00

x 10^4
9
8
7
6
5
4
3
2
1
0



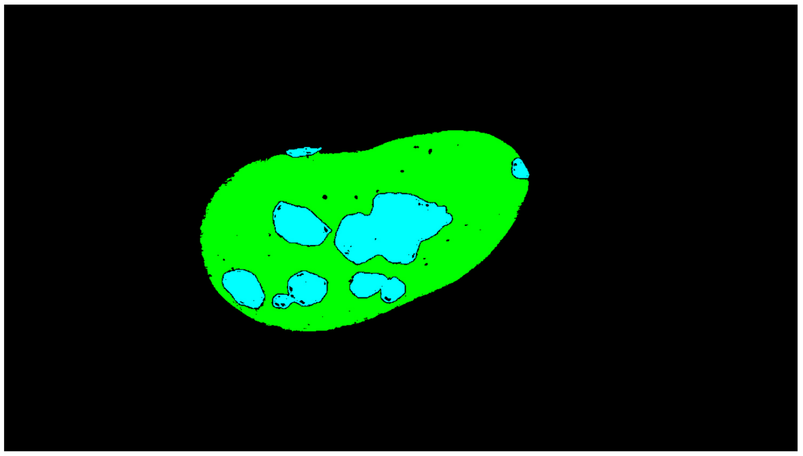
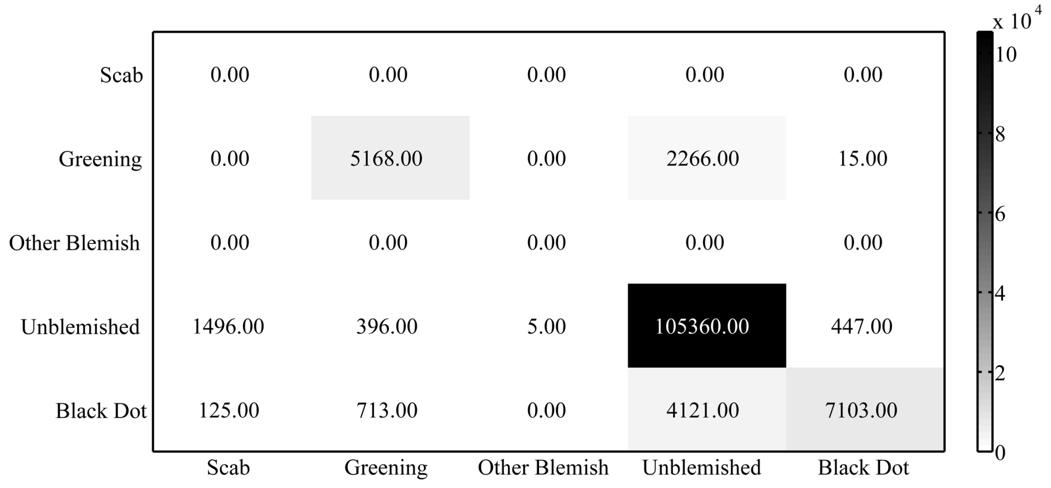
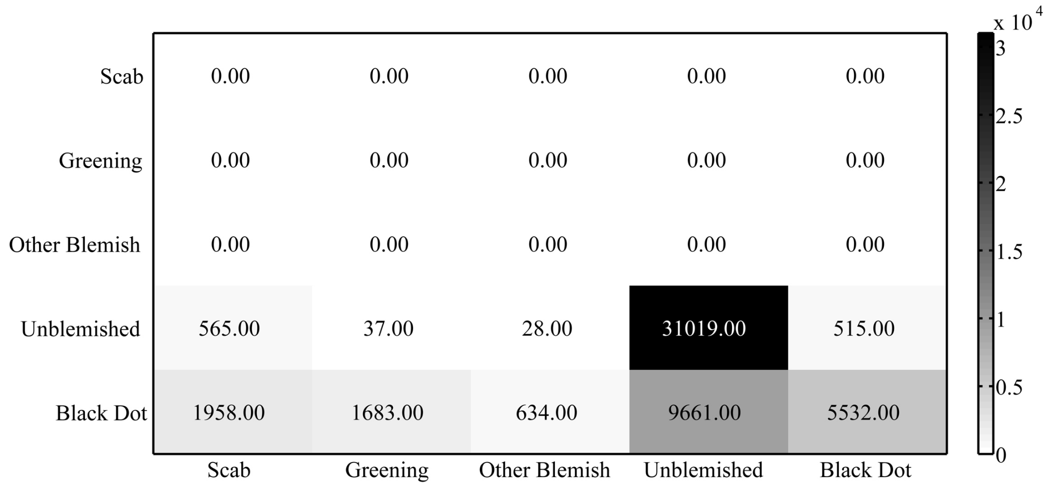


Figure 3: Top: Confusion matrix for the second input image; Middle: Classification results produced by the QA-TADD system; Bottom: Ground truth



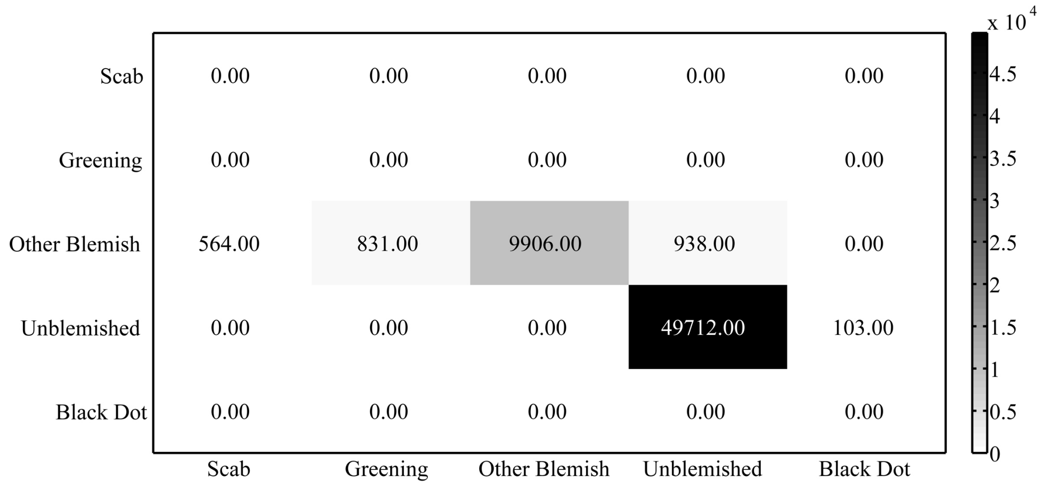
	Scab	Greening	Other Blemish	Unblemished	Black Dot
Scab	0.00	0.00	0.00	0.00	0.00
Greening	0.00	5168.00	0.00	2266.00	15.00
Other Blemish	0.00	0.00	0.00	0.00	0.00
Unblemished	1496.00	396.00	5.00	105360.00	447.00
Black Dot	125.00	713.00	0.00	4121.00	7103.00

Figure 4: Confusion matrix for the seventh input image



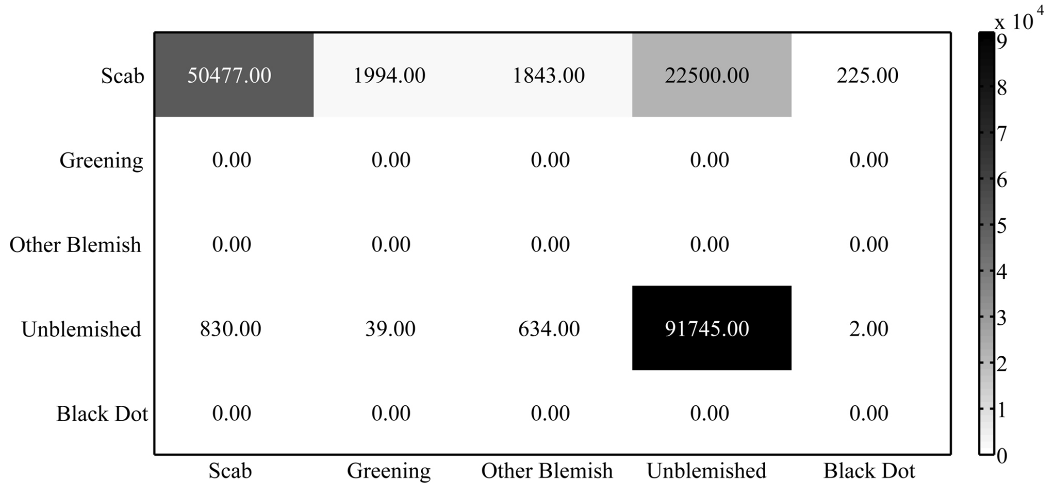
	Scab	Greening	Other Blemish	Unblemished	Black Dot
Scab	0.00	0.00	0.00	0.00	0.00
Greening	0.00	0.00	0.00	0.00	0.00
Other Blemish	0.00	0.00	0.00	0.00	0.00
Unblemished	565.00	37.00	28.00	31019.00	515.00
Black Dot	1958.00	1683.00	634.00	9661.00	5532.00

Figure 5: Confusion matrix for the eleventh input image



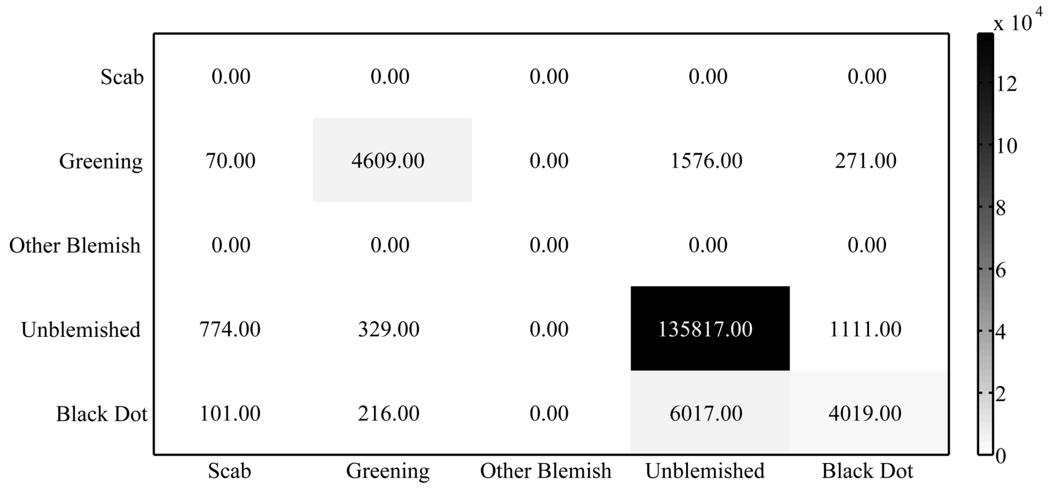
	Scab	Greening	Other Blemish	Unblemished	Black Dot
Scab	0.00	0.00	0.00	0.00	0.00
Greening	0.00	0.00	0.00	0.00	0.00
Other Blemish	564.00	831.00	9906.00	938.00	0.00
Unblemished	0.00	0.00	0.00	49712.00	103.00
Black Dot	0.00	0.00	0.00	0.00	0.00

Figure 6: Confusion matrix for the twelfth input image



	Scab	Greening	Other Blemish	Unblemished	Black Dot
Scab	50477.00	1994.00	1843.00	22500.00	225.00
Greening	0.00	0.00	0.00	0.00	0.00
Other Blemish	0.00	0.00	0.00	0.00	0.00
Unblemished	830.00	39.00	634.00	91745.00	2.00
Black Dot	0.00	0.00	0.00	0.00	0.00

Figure 7: Confusion matrix for the twenty-first input image



	Scab	Greening	Other Blemish	Unblemished	Black Dot
Scab	0.00	0.00	0.00	0.00	0.00
Greening	70.00	4609.00	0.00	1576.00	271.00
Other Blemish	0.00	0.00	0.00	0.00	0.00
Unblemished	774.00	329.00	0.00	135817.00	1111.00
Black Dot	101.00	216.00	0.00	6017.00	4019.00

Figure 8: Confusion matrix for the thirty-fourth input image