Performance Characterization in Computer Vision
A Tutorial

Adrian F. Clark and Christine Clark
VASE Laboratory, Electronic Systems Engineering
University of Essex, Colchester, CO4 3SQ, UK
{alien,cc}@essex.ac.uk

This document provides a tutorial on performance characterization in computer vision. It explains why learning to characterize the performances of vision techniques is crucial to the discipline's development. It describes the usual procedure for evaluating vision algorithms and its statistical basis. The use of a software tool, a so-called test harness, for performing such evaluations is described. The approach is illustrated on an example technique.

Contents

This tutorial was developed under the ægis of a project called *Performance Characterization in Computer Vision* (project 1999-14159, funded under the European Union's IST programme). See http://peipa.essex.ac.uk/benchmark/ for further information.

1 Introduction

The discipline variously known as *Computer Vision*, *Machine Vision* and *Image Analysis* has its origins in the early artificial intelligence research of the late 1950s and early 1960s. Hence, roughly two generations of researchers have pitted their wits against the problem. The pioneers of the first generation worked with computers that were barely capable of handling image data — processing had to be done line-by-line from backing store — and programs almost always had to be run as batch jobs, ruling out any form of interaction. Even capturing digital images was an impressive feat. Under such difficult conditions, the techniques that were developed were inevitably based on the mathematics of image formation and exploited the values of pixels in neighbouring regions. Implementing them was a non-trivial task, so much so that pretty well any result was an impressive achievement.

The second generation of researchers coincided with the birth of the workstation. At last, an individual researcher could process images online, display them, and interact with them. These extra capabilities allowed researchers to develop algorithms that involved significant amounts of processing. A major characteristic of many algorithms developed during this second generation was the quest for optimality. By formulating and manipulating a set of equations that described the nature of the problem, a solution can usually be obtained by a least-squares method which, of course, is in some sense optimal. Consequently, any number of techniques appeared with this ‘optimality’ tag. Sadly, none of these papers were able to provide credible *experimental* evidence that the results from the optimal technique was significantly better than existing (presumably sub-optimal) ones.

We are now in the early years of the third generation. Computers, even PCs, are so fast and so well-endowed with storage that it is entirely feasible to process large datasets of images in a reasonable time — and this means it is possible to quantify the performance of an algorithm. As a result, the vision community has finally started to turn its attention to issues related to testing and comparing algorithms: performance assessment. The most visible (no pun intended) aspect of this is the competitions that are often organized in association with major vision conferences. These essentially ask the question “which algorithm is best?” Although a natural enough question to ask, it lacks subtlety and is potentially rather dangerous: if the community as a whole adopts an algorithm as “the standard” and concentrates on improving it further, that action can stifle research into other algorithms.

A better approach is to make available a “strawman” algorithm which embodies an approach that is known to work but does not represent the state of the art. This might be, for example, the “eigenfaces” approach [1] without refinements for face recognition, the Canny edge detector, and so on. Authors can use the strawman for comparison, and anything that out-performs it is a good candidate for publication; conversely, anything that performs less well than the strawman needs improvement.

If asking which algorithm is best is unsubtle, then what is a more appropriate question? We believe researchers should be asking “why does one algorithm out-perform another?” To answer the latter question, one must explore what characteristics of the inputs affect the algorithms’ performances and by how much. In fact, one can carry this process out on an algorithm in isolation as well as comparing algorithms. This is what is meant by *performance characterization*, the subject of this tutorial, and is a closer match to the way knowledge and understanding are advanced in other areas of science and engineering.

It may seem from the above that performance assessment and characterization are intellectual exercises, divorced from the gritty realities of applying vision techniques to real-world problems; but nothing could be further from the truth. Vision techniques have a well-deserved reputation for being fragile, working well for one developer but failing dismally for another who applies them to imagery with slightly different properties. This should not come as a surprise, for very few researchers have made any effort to assess how well different algorithms work on imagery as its properties differ — say as the amount of noise present changes — never mind making the algorithms more robust to them. So, far from being an abstract exercise, performance characterization is *absolutely essential* if computer vision is to escape from the research laboratory and be applied to the thousands of problems that would benefit from it.

The process conventionally adopted for performance assessment and characterization has not yet been expounded; that is done in Section 2. Section 3 and Section 4 then describe the underlying statistical principles and describes those statistical tests, displays and graphs in common use for characterizing an individual algorithm and for comparing algorithms respectively. As testing is an onerous task when carried out manually, Section 5 describes a software tool that can be used to automate much of the work. These tests and tools are put to good work in Section 6, which shows how a simple image analysis technique can be characterized. Finally, Section 7 gives some concluding remarks.

2 The Performance Assessment and Characterization Processes

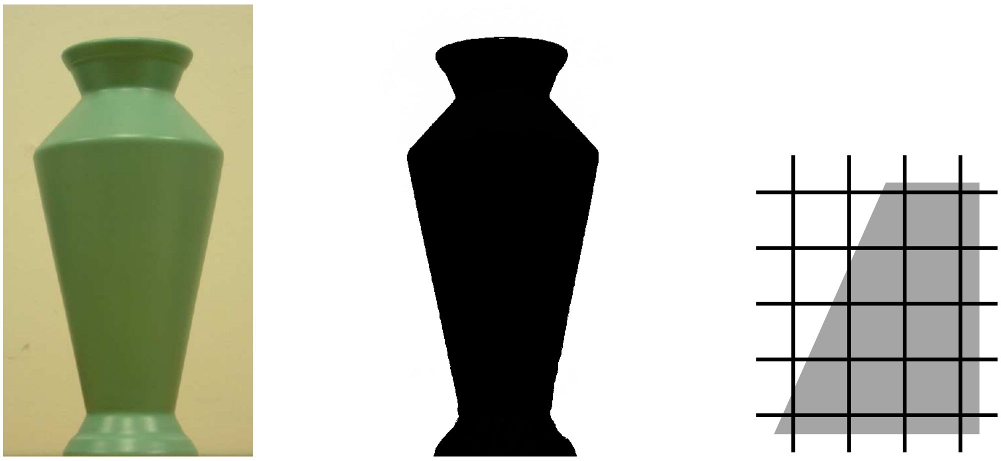
There are few occasions when it is possible to predict the performance of an algorithm analytically: there are normally too many underlying assumptions, or the task is just too complicated (but see [2] for a rare exception). So performance is almost universally assessed empirically, by running the program on a large set of input data whose correct outputs are known and counting the number of cases in which the program produces correct and incorrect results.

Each individual test that is performed can yield one of four possible results:

True positive: (also known as *true acceptance* or *true match*) occurs when a test that should yield a correct result does so.

True negative: (also known as *true rejection* or *true non-match*) occurs when a test that should yield an incorrect result does so.

False negative: (also known as *false rejection*, *false non-match* or *type I error*) occurs when a test that should yield a correct result actually yields an



(a) Original image

(b) Corresponding template

(c) Ambiguity at edge of template

Figure 1: An object against a plain background

incorrect one.

False positive: (also known as *false acceptance*, *false match*, *false alarm* or *type II error*) occurs when a test that should yield an incorrect result actually yields a correct one.

There is occasionally some confusion in the literature over the terms “false negative” and “false positive,” which is why their meanings have been given here. “False negative,” for example, can be thought of as a case in which a test should have given a true negative but failed to do so. The testing procedure involves keeping track of these four quantities. Performance assessment work normally uses them with little additional consideration: the algorithm with the highest true rate (or, equivalently, the lowest false rate) is normally taken in comparisons and competitions to be the best.

To be able to perform testing in this way, each individual test requires three pieces of information:

1. the input to the program under test;
2. the corresponding expected output from the program under test;
3. whether this output corresponds to a success or a failure.

Vision researchers rarely test explicitly for failures, *e.g.* by running a vision algorithm on an image whose pixels are all set to the same value.

To fix these ideas in our minds, let us consider the example of using a procedure to try detecting in an image an object surrounded by a plain background. Specifically, Figure 1(a) shows a vase against a plain background. A template image, Figure 1(b), can be constructed to determine which pixels are to be regarded as ‘vase’ pixels; the rest would be regarded as ‘background’. On applying the procedure, if a pixel is classed as 'vase' and it is known from the template to be part of the vase, then this pixel is a *true positive*. If a pixel is classed as 'background' and it is known to be part of the background (*i.e.*, not vase), then this pixel is a *true negative*. If a pixel is classed as 'vase' but it is known to be part of the background, then this pixel is a *false positive*. If a pixel is classed as 'background' but it is known to be part of the vase, then this pixel is a *false negative*.

While it is obvious that the performance depends on how accurately the template has been determined, these values give a measure of algorithm performance. In particular, we should expect both false positives and false negatives to occur most frequently in the region where the object meets the background because there will be pixels where there are contributions from both the object and its background, as illustrated in Figure 1(c). It would be wise to weight errors in this region less than errors elsewhere in the image.

It must be appreciated that there is always a trade-off between true positive and false positive detection. If a procedure is set to detect all the true positive cases then it will also tend to give a larger number of false positives. Conversely, if the procedure is set to minimize false positive detection then the number of true positives it detects will be greatly reduced. However, tables of true positives *etc.* are difficult to analyze and compare, so results are frequently shown graphically using ROC or DET curves (see Section 3).

It should in principle be possible to compare the success rates of algorithms obtained using different datasets; but in practice this does not work. This is, in effect, the same as saying that the datasets used in performing the evaluations are not large and comprehensive enough, for if they were it *would* be possible simply to compare success rates. The number of ways in which image data may vary is probably so large that it is not feasible to encompass all of them in a dataset, so it is currently necessary to use the same datasets when evaluating algorithms — and that means using the same training data as well as the same test data. Sadly, little effort has been expended on the production of standard datasets for testing vision algorithms until recently; the FERET dataset (*e.g.*, [3]) is probably the best example to date.

When the performances of algorithms are compared, it is not enough simply to see which has the better success rate, for this takes no account of the number of tests that has been performed: the size of the dataset may be sufficiently small that any difference in performance could have arisen purely by chance. Instead, a standard statistical test, M^{c}Nemar's test (see Section 4), should be used as it takes this into account. M^{c}Nemar's test requires that the results of applying both algorithms on the same dataset are available, so this fits in well with the comments in the previous paragraph.

An argument that is often put forward is that vision algorithms are designed to perform particular tasks, so it only makes sense to test an algorithm on data relating precisely to the problem, *i.e.* on real rather than simulated imagery. While this is true to a certain extent — the range of applications of vision tasks is indeed vast — it ignores the fact that there are generic algorithms that underlie practically all problem-specific techniques, *e.g.* edge detection. Indeed, this really illustrates the distinction between two different types of testing:

technology evaluation: the response of an algorithm to factors such as adjustment of its tuning parameters, noisy input data, *etc.*;

application evaluation: how well an algorithm performs a particular task;

where the terminology has been adapted from [4]. Technology evaluation is one example of performance characterization.

To illustrate the distinction between technology and application evaluation, let us consider an example that will be familiar to most computer vision researchers, namely John Canny's edge detector [5]. Technology evaluation involves identifying any underlying assumptions (*e.g.,* additive noise) and assessing the effects of varying its tuning parameters (*e.g.,* its thresholds, the size of its Gaussian convolution mask). This is best done using *simulated* data, as it provides the only way that all characteristics of the data can be known. Conversely, application evaluation assesses the effectiveness of the technique for a particular task, such as locating line-segments in fMRI datasets. This second task must, of course, be performed using real data. If the former is performed well, the researcher will have some idea of how well the algorithm is likely to perform on the latter simply by estimating the characteristics of the fMRI data — how much and what type of noise, and so on.

3 Assessing an Individual Algorithm

Tables of true positives *etc.* are difficult to analyse and compare. Hence, researchers have introduced methods of presenting the data graphically. We shall consider two of these, the *receiver operating characteristic* (ROC) curve and the *detection error trade-off* (DET) curve. We shall also consider a display that is frequently used in describing the performance of classification studies, namely the *confusion matrix*. Other measures and displays do exist, of course; many of them are described in [6].

3.1 The Receiver Operating Characteristic Curve

A ROC curve is a plot of false positive rate against true positive rate as some parameter is varied. ROC curves were developed to assess the performance of radar operators during the second World War. These operators had to make the distinction between friend or foe targets, and also between targets and noise, from the blips they saw on their screens. Their ability to make these vital distinctions was called the receiver operating characteristic. These curves were taken up by the medical profession in the 1970s, who found them useful in bringing out the *sensitivity* (true positive rate) versus *specifity* ($1-$false positive rate) of, for example, diagnosis trials. ROC curves are as interpreted as follows (see Figure 2):

- the closer the curve approaches the top left-hand corner of the plot, the more accurate the test;

- the closer the curve is to a 45° diagonal, the worse the test;

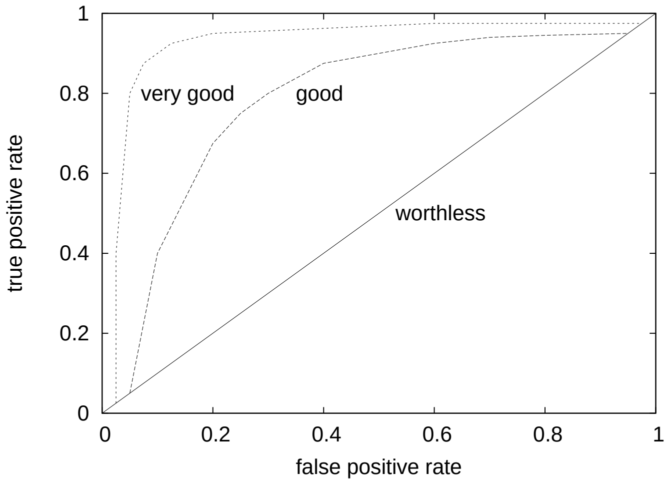


Figure 2: Examples of ROC Curves

- the area under the curve is a measure of the accuracy of the test;
- the plot highlights the trade-off between the true positive rate and the false positive rate: an increase in true positive rate is accompanied by an increase in false positive rate.

It should be noted that there does not appear to be a convention as to the orientation of the plot, so one encounters a variety of orientations in the literature; in such cases, the above interpretation must be adjusted accordingly.

Figure 2 shows ROC curves for a very good, a good and a very poor (worthless) test. As stated above, the area under each curve gives a measure of accuracy. An area of unity represents a perfect test, while a measure of 0.5 (*e.g.*, a 45° diagonal) represents a failed test (random performance). Various methods of estimating the area under the curves have been suggested, including using a maximum likelihood estimator to fit the data points to a smooth curve, using Simpson's rule, and fitting trapezoids under the curve. There are, however, more effective ways of assessing the overall accuracy of an algorithm, as we shall see.

Error considerations can be indicated on these plots. For example, if a single test is run on many different sets of images, then the mean false-positive rate can be plotted against the mean true-positive rate. The assessed confidence limits can then be plotted as error bars or error ellipses around the points.

3.2 The Detection Error Trade-off Curve

A DET curve is a plot of false positive rate versus false negative rate and thus gives equal emphasis to both types of error; see Figure 3. The plot usually has logarithmic scales on both axes, so DET curves tend to be more spread out than ROC curves, making it easier to distinguish individual algorithms' results. The

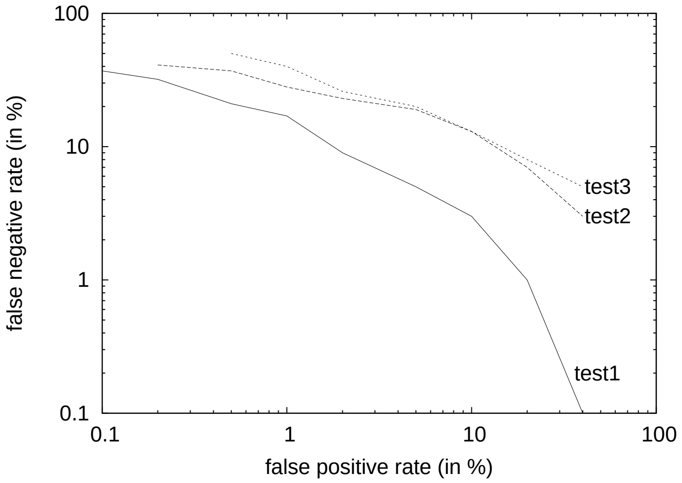


Figure 3: Examples of DET Curves

curves also tend to be close to linear. DET curves can be used to plot matching error rates and decision error rates as well as confidence intervals/boxes. Figure 3 shows an example of a DET curve plot for three tests.

The DET curve plot highlights the trade-off between the false-positive and false-negative rates, which is useful in areas where trade-offs between the two error types are important. If a 'curve' is a straight line then this shows that the underlying likelihood distributions from the procedure are Normal: a bell-shaped curve plotted on linear axes results in a straight line when plotted on logarithmic axes. Also, the diagonal $y = -x$ on the Normal deviate scale (*i.e.*, plotted on linear axes) represents a failed test (random performance).

Some researchers refer to the equal error rate (EER) of a particular test. The EER is the point at which the false positive rate is equal to the false negative rate. This may be of use in applications where the cost of each type of error is equal. The smaller the EER, the better. However, in general the whole DET curve is considered.

We have seen that ROC and DET curves are useful in assessing how different parameters applied to an algorithm affect performance. The following section describes how algorithms can be compared.

3.3 Confusion Matrices

A confusion matrix [7] contains information on the actual and predicted classifications performed by a system. For example, for the digit-recognition task described in Section 6, a confusion matrix like that in Table 1 might arise.

Numbers along the leading diagonal of the table represent digits that have been classified correctly, while off-diagonal values show the number of mis-classifications. Hence, small numbers along the leading diagonal show cases in which classification performance has been poor, as with '8' in the table. Here, the actual digit '0' has been mis-classified as '8' ten times and as '6' once, while

actual	predicted									
	0	1	2	3	4	5	6	7	8	9
0	20	0	0	6	0	0	1	0	10	0
1	0	25	0	0	0	0	0	6	0	0
2	0	0	31	0	0	0	0	0	0	0
3	0	0	21	0	0	0	0	0	10	0
4	0	0	0	31	0	0	0	0	0	0
5	0	0	0	0	0	22	0	0	9	0
6	1	0	0	1	0	2	23	0	3	1
7	0	8	0	0	0	0	0	23	0	0
8	4	0	1	3	2	1	3	0	13	4
9	0	0	0	2	0	0	0	3	1	27

Table 1: Confusion Matrix for a Digit Recognition Task

the digit '1' mis-classified as '7' six times. Conversely, the digit '2' has never been mis-classified. There is no reason, of course, why the matrix should be symmetric.

In the particular case that there are two classes, success and failure, the confusion matrix just reports the number of true positives, *etc.* as shown below.

	predicted negative	predicted positive
actual negative	TN	FP
actual positive	FN	TP

4 Comparing Algorithms

4.1 Using ROC and DET curves

The most common way that algorithms are compared in the literature is by means of their ROC or DET curves. This is acceptable to some extent; but the problem is that researchers hardly ever indicate the accuracy of the points in the curve using error bars or equivalent, and hence one cannot tell whether any differences in performance are significant.

ROC curves tend not to be as straightforward as those shown in Figure 2. Often the curves to be compared cross each other, and then it is up to the user to decide which curve represents the best method for their application. For example, Figure 4, shows that `alg1` may be superior to `alg2` when a high true-positive rate is required but `alg2` may be preferred when a low false-positive rate is required.

As the accuracy of vision algorithms tends to be highly data-dependent, comparisons of curves obtained using different data sets should be treated with suspicion. Hence, the only viable way to compare algorithms is to run them

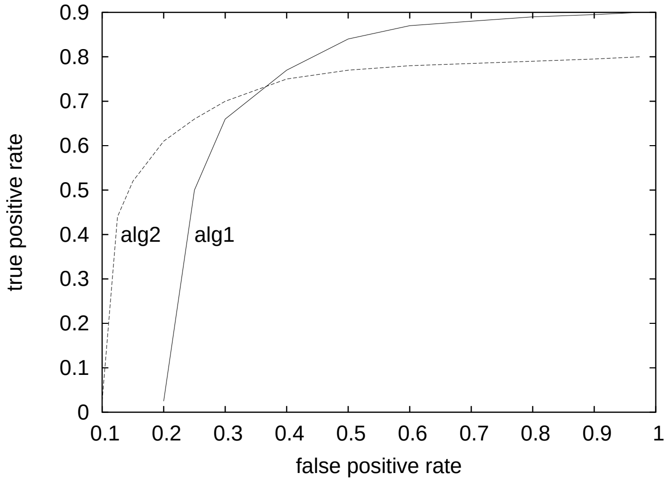


Figure 4: Crossing ROC Curves

on the same data. In principle, one could generate ROC or DET curves for any number of algorithms, plot them with error bars, and perform visual comparisons. Even in this case, however, it is usually difficult to be sure whether one algorithm out-performs another significantly.

Hence, comparisons of algorithms tend to be performed with a specific set of tuning parameter values. (Running them with settings that correspond to the equal error rate is probably the most sensible.) When this is done, perhaps under the control of a test harness such as the one described in Section 5, an appropriate statistical test can be employed. This must take into account not only the number of false positives *etc.* but also the number of tests: if one algorithm obtains 50 more false positives than another in 100,000 tests, the difference is not likely to be significant; but the same difference in 100 tests almost certainly is.

4.2 M$^{\text{c}}$Nemar's Test

The appropriate test to employ for this type of comparison is M$^{\text{c}}$Nemar's test. This is a form of chi-square test for matched paired data. Consider the following 2×2 table of results for two algorithms:

	Algorithm A Failed	Algorithm A Succeeded
Algorithm B Failed	N_{ff}	N_{sf}
Algorithm B Succeeded	N_{fs}	N_{ss}

M$^{\text{c}}$Nemar's test is:

$$\chi^2 = \frac{(|N_{sf} - N_{fs}| - 1)^2}{(N_{sf} + N_{fs})} \tag{1}$$

Z value	Degree of confidence Two-tailed prediction	Degree of confidence One-tailed prediction
1.645	90%	95%
1.960	95%	97.5%
2.326	98%	99%
2.576	99%	99.5%

Table 2: Converting Z Scores onto Confidence Limits

where the -1 is a continuity correction. We see that M^{c}Nemar's test employs both false positives and false negatives, rather than just one of them.

If the number of tests is greater than about 30 then central limit theorem applies. The central limit theorem states that if the sample size is moderately large and the sampling fraction is small to moderate, then the distribution is approximately Normal. In such a case, the Z score (standard score) is obtained from (1) as:

$$Z = \frac{(|N_{sf} - N_{fs}| - 1)}{\sqrt{N_{sf} + N_{fs}}} \tag{2}$$

If Algorithm A and Algorithm B give very similar results then Z will tend to $Z = 0$. As their results, diverge Z will increase. Confidence limits can be associated with the Z value as shown in Table 2. Values for two-tailed and one-tailed predictions are shown in the table as either may be needed depending on the hypothesis used: if we assessing whether two algorithms differ, a two-tailed test should be used; and if we are determining whether one algorithm is better than another, a one-tailed test should be used.

Further information can also be gleaned from N_{sf} and N_{fs}: if these values are both large, then tends to Algorithm A succeed where Algorithm B fails and *vice versa*. This is valuable to know, as we can devise a new algorithm that uses both in parallel and takes the value of Algorithm B where Algorithm A fails, and *vice versa* — this should yield an overall improvement in accuracy. This is actually a rather significant statement with regard to the design of vision systems: rather than combining the results from algorithms in the rather *ad hoc* manner that usually takes place, M^{c}Nemar's test provides a principled approach that tells us not only *how to do it* but also *when it is appropriate to do it* on the basis of technology evaluation — in other words, technology evaluation needs to be an inherent part of the algorithm design process.

5 Automating the Testing Process

As performance characterization involves running an algorithm on hundreds, and perhaps even thousands, of test cases, it is obviously not feasible to carry it out by hand. Consequently, individual researchers tend to write software to automate the process, often in the form of a short "script" for one of the Unix shells, or in the Perl, Python or Tcl programming languages. Such scripts typically do little more than execute the program, saving the result in a file for separate analysis. If one is performing such tests in isolation, this is fine; but when the test results are to be used for comparison, perhaps as part of a competition, then care is needed. Although competitions frequently (though not always) specify a text "protocol," such protocols are described in words, and there are inevitable differences in ambiguities and interpretation, such as the "false positive" interpretation alluded to above. Hence, the best way to perform testing, especially when the results are to be used for comparison, is to use specially-designed software: a "test harness."

The problem with a test harness for performance characterization is that it must be possible for researchers spread around the world to use it with their own software, and that is difficult to achieve. There are two basic ways in which it can be achieved:

- researchers upload their software to a central site, where it is executed and the results made available;

- researchers download the test harness and use it locally.

The first of these is certainly easier and has the advantage that comparisons may include execution time, memory usage *etc.* in addition to performance. It also avoids problems in releasing to the public datasets that may be sensitive (*e.g.*, medical imagery). An excellent example of this approach is Algoval [8], though that is restricted to class libraries written in Java. The largest problem with this approach is that researchers may not be happy for their software, even in compiled form, to be "given out." This is especially true for industrial researchers and out-weighs the technical advantages of the approach. As a result, distributed testing, in which researchers execute the tests on their own computers, is likely to gain more acceptance in the vision research community. The particular test harness that we shall consider here is HATE [9].

5.1 The HATE Test Harness

HATE is an acronym for *Harness for algorithm Testing and Exploration*, though its name perhaps describes more accurately the emotion people feel when faced with having to perform tests. It was designed with the requirements of distributed performance characterization in mind and, as a result, has several unique features:

- it allows tests to be specified independently of the software being tested in a *test script*;

- it allows software written in any programming language to be used with the harness by means of an *interface script*;

- it is Internet-friendly, able to download test scripts and datasets as required.

This separation of the interface to the program under test, the test script, and the harness itself means that they can be written by different people — which is both desirable and what happens in practice. HATE is written in Perl, so both interface and test scripts must also be written in Perl; however, interface scripts are typically only a few lines, while test scripts are straightforward and often generated by software. The algorithm developer writes the interface script, so he or she can use his or her algorithm with HATE. The test developer writes the test script independently of any particular piece of software that may be tested, effectively expressing the test protocol as software. The test harness provides the software "glue" that brings these parts together; and, as it works out what is a true positive *etc.*, there is a consistent interpretation of terminology.

HATE can, of course, be used with test scripts written locally. Its true power, however, is when it is used with scripts that are designed specifically for characterizing and comparing algorithms. Such scripts are made available at a central site (its URL is built into HATE) and can be downloaded and executed simply by specifying their name. HATE will also download data files needed by the test if they are not already available locally.

In its most straightforward form, HATE executes the series of tests specified in the test script and reports the number of true positives *etc.* obtained from the program under test. However, as we have seen, such information is of limited value. So HATE can be told to generate data for plotting ROC or DET curves (in forms compatible with Gnuplot [10], R [11] and most spreadsheets). Indeed, HATE can systematically vary not just one but any number of parameters, though plotting the result may prove difficult.

HATE can also generate a "transcript" in which it reports the result of each individual test in an easy-to-parse form. Transcript files obtained from different algorithms can then be compared, and such comparisons are performed in a statistically-valid way using M^{c}Nemar's test, and the output tells the researcher whether any difference in success or failure rates is statistically significant.

6 An Illustrative Example

The easiest to see how HATE works is to show how it is used. To do so, let us consider the classification of images of handwritten digits. One of the simplest algorithms for this kind of task is WISARD. An outline of the algorithm and some features of the particular implementation we shall consider are given in Appendix A. (The software may be downloaded from the HATE web-site.) Assuming HATE is installed on your computer, which is connected to the Internet, you find out whether there is an appropriate test already available by typing the command

```
hate -mode list-tests
```

This tells HATE to give the name and a one-line synopsis for each test that it knows about, either locally or on the central HATE web-site. You will find there is a script called `hdigits.hate` for precisely the kind of algorithm you have developed. You then type the command

```
hate -mode describe hdigits
```

to receive a description of the test. (HATE downloads the test script and caches it for subsequent use.) HATE outputs a description the dataset that should be used for training your algorithm, so you download it and train WISARD appropriately. The final step of preparation is to write an interface routine, stored in the file `interface.pl` in the directory in which you intend to perform the tests.

The series of tests described in the script is executed by the command

```
  hate hdigits
```

which results in the following output

```
#     tests        TP        TN        FP        FN
       1434       450       320       300       364
```

Here, `TP` stands for "true positive" *etc.*, so for 1434 tests, 450 of them resulted in true positives, *etc.*.

To generate the data for an ROC curve, one would type the command

```
  hate -mode roc -param "threshold=0,1,2,3,4" hdigits
```

which causes the series of tests to be performed several times, each with the value of some symbol called `threshold` being set to the successive values listed. (The value of a symbol defined on the command line can be picked up in the interface script.) In the context of WISARD, the threshold is the amount by which the highest score must exceed all other scores on a particular test in order to be deemed a success. The resulting output is

```
#     tests        TP        TN        FP        FN
       1434       450       320       300       364
       1434       140       100       594       600
       1434        15        14       700       705
       1434         1         1       706       726
       1434         0         0       706       728
```

which can be fed into `Gnuplot`, for example, to produce an ROC or DET curve.

The information from a single run of HATE, or an ROC curve produced by varying a parameter, is useful during the algorithm development process as one can see in a few minutes whether or not an "improvement" actually does improve performance. Similarly, we would like to see if an algorithm under development out-performs existing ones developed by other researchers. Although one can do this manually, a far better way is to use HATE to perform the comparison.

Let us imagine there are two other digit-classification algorithms, WARLOCK and WITCH, with which we would like to compare WISARD. To do this, one first tells HATE to record a "transcript" during execution, saving the output in a file.

```
  hate -format transcript hdigits > wisard.out
```

Similar command lines are used to produce transcripts from WARLOCK and WITCH. The three transcripts are then compared with a single command

```
hate -mode compare wisard.out warlock.out witch.out
```

As with other files, `warlock.out` and `witch.out` may be downloaded over the Internet by HATE if they do not exist locally. HATE checks that the same version of the same test script was used in all cases; if it does, it performs case-by-case comparisons and produces a series of tables in textual or LaTeX format; the specific output for the above comparison is listed in Figure 5.

The first table summarizes how well each algorithm performed. In this example WISARD gave better results than WITCH in 1254 cases but was only better than WARLOCK in 130 cases. WITCH was better than WISARD and WARLOCK in 180 cases. In no case was WARLOCK better than WISARD but it did manage to be better than witch in 1124 cases.

Following this are tables where pairs of algorithms are compared in the style described in Section 4. In this example, the first pair to be compared are WISARD and WITCH. We can see that in no case did both WISARD and WITCH fail for the same test. WISARD succeeded in 1254 cases where WITCH failed while WITCH succeeded in 180 cases where WISARD failed. In no case did both WITCH and WISARD succeed. Tables for comparison of WISARD and WARLOCK and WITCH and WARLOCK are then given.

If we consider the 2×2 table for the comparison of WISARD and WITCH then:

$$Z = \frac{(|1254 - 180| - 1)}{\sqrt{1254 + 180}} = 28.34$$

We can look up this Z score in Table 2 to find the associated confidence limit. As $28.34 > 2.576$ we can say with more than 99% confidence that algorithm A (WISARD) and algorithm B (WITCH) do not give equivalent results. As algorithm A (WISARD) gave a larger number we can say with more than 99.5% confidence that algorithm A (WISARD) was indeed superior to algorithm B (WITCH) for these tests.

7 Concluding Remarks

This document has given an overview of performance characterization. It started with a discussion of why this topic has received little attention by the computer vision research community to date and explains that more effort must be put into it if the discipline is to mature. The motivation for carrying out performance characterization studies should hence be clear.

The document goes on to describe how performance is currently assessed, by accumulating performance against a set of test data, and distinguished between technology evaluation, the performance of an algorithm in isolation, and application evaluation, the performance of an algorithm on a specific type of data with a particular problem in mind. Those measures and graphics most commonly used to display performance results were then presented, both for an individual algorithm and for the comparison of algorithms.

The use of test harnesses for taking the hard work out of performance studies was then considered and the benefits, not all of which are obvious, noted.

		WORSE		
		`wisard.out`	`witch.out`	`warlock.out`
BETTER	`wisard.out`	xxxxxxxxxx	1254	130
	`witch.out`	180	xxxxxxxxxx	180
	`warlock.out`	0	1124	xxxxxxxxxx

	`wisard.out` failed	`wisard.out` succeeded
`witch.out` failed	0	1254
`witch.out` succeeded	180	0

Z score is 28.335
99% confident that `wisard.out` and `witch.out` do not give equivalent results
99.5% confident that `wisard.out` was superior to `witch.out`

	`wisard.out` failed	`wisard.out` succeeded
`warlock.out` failed	180	130
`warlock.out` succeeded	0	1124

Z score is 11.314
99% confident that `wisard.out` and `warlock.out` do not give equivalent results
99.5% confident that `wisard.out` was superior to `warlock.out`

	`witch.out` failed	`witch.out` succeeded
`warlock.out` failed	130	180
`warlock.out` succeeded	1124	0

Z score is 26.114
99% confident that `witch.out` and `warlock.out` do not give equivalent results
99.5% confident that `warlock.out` was superior to `witch.out`

Figure 5: Example Output from HATE in Compare Mode

The use of the HATE test harness was then described in assessing a simple vision algorithm and comparing it with two others. This can reduce the time required literally to ten minutes and represents the state of the testing art at the time of writing.

We are a long way from solving all the problems associated with performance characterization. The major technical problems lie in automating the production of histograms that show how algorithms react to changes in the properties of their inputs, and in optimizing parameter settings for an algorithm. Both topics are being worked on by the developers of the HATE test harness and will appear in due course.

Characterizing the performance of algorithms is now feasible. The characterization may not be perfect but will undoubtedly be a vast improvement on having no idea how algorithms react to the properties of the data presented to them. A valuable resource would be a "data sheet" for popular algorithms, somewhat analogous to the data books that describe electronic components. A major problem then is in devising a methodology that describes how algorithms can be joined together in a way that is statistically valid. A start to this problem has been made [4] but there is far to go.

However, by far the greatest short-term problem is in convincing the computer vision community as a whole of the importance of evaluation and characterization. We encourage you to help do so, and there are several ways you can contribute:

- use existing test scripts when evaluating your algorithms, and say so in publications;
- make available transcript files of the results of test for your algorithms;
- make the code and interface file for your algorithms available;
- if there isn't a test script for your particular research, make available one and its accompanying data;
- encourage others to participate in performance studies too.

Together we can really crack the vision problem.

A The WISARD Neural Network

WISARD is a simple pattern recognition scheme, devised in the 1970s by Wilkie, Stonham and Aleksander; the name stands for *Wilkie, Stonham and Aleksander's Recognition Device* [12]. It is usually described as a neural network, though some do not regard so as it is weightless and can be trained by a single data presentation. It is occasionally described as an "associative memory," which is closer to how it actually works, and as an "n-tuple" network. However, we don't need to get ourselves bogged down in such pedantry here; WISARD is an approach to pattern recognition and there is an implementation of it that needs to be examined and assessed.

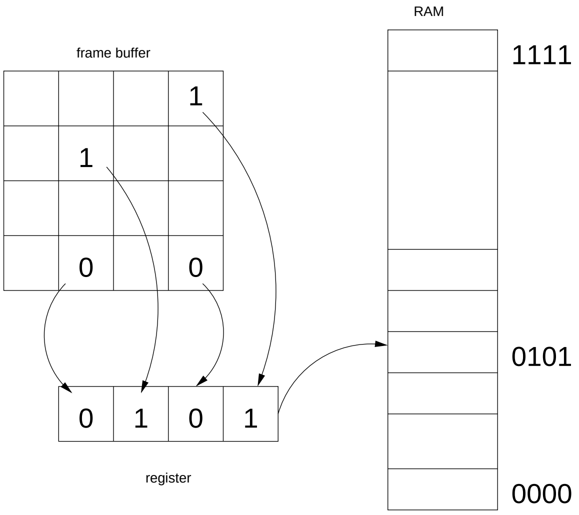


Figure 6: Illustration of the Operation of a WISARD Network

The operation of WISARD is illustrated in Figure 6. The system consists of a frame buffer, a register or *discriminator element* and a 1-bit RAM. (The simplicity of the architecture is because WISARD was designed to be implemented in hardware.) WISARD is intended to operate on binary images, *i.e.*ones whose pixels may take only the values zero and unity.

Initially, every element of the RAM is cleared (set to zero). Several pixel locations, four in Figure 6, are chosen at random and connected to the bits of the register. When an image is loaded into the frame buffer, the values held in those four locations determine the value in the register. What happens next depends on whether WISARD is being trained or tested (used for recognition):

- when being trained, the value in the RAM addressed by the register is set;
- when being tested, the value in the RAM addressed by the register is compared with unity.

Fairly obviously, when the test pattern is identical to the training pattern, at least in the selected locations, a match will be found.

Of course, if WISARD used only four samples from an image, it would not be particularly robust; hence, several other sets of four randomly-chosen pixels are used in the same way. The number of comparisons from groups of four pixels that produce a match are added up, resulting in a "score" that determines how close the overall match is.

The particular implementation that we shall consider is used in one of two ways:

Training. To train a network you execute the program as follows:

```
wisard train wiz1.net <data>train/1-*.ppm
```

Here, `train` is a keyword that tells the program to run in training mode, `wiz1.net` is the file in which the trained network is saved, and the files on the remainder of the command line are used for training.

Testing. To test a network you execute the program as follows:

```
wisard test wiz1.net <data>test/1-*.ppm
```

Here, `test` is a keyword that tells the program to run in testing mode, `wiz1.net` is the file from which a trained network is loaded, and the files on the remainder of the command line are used for testing.

In both cases, `<data>` is the top of the directory tree in which the training and test data are kept.

References

[1] M. Turk and A. Pentland. Eigen faces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[2] S. J. Maybank. Probabilistic analysis of the application of the cross ratio to model-based vision. *International. Journal of Computer Vision*, 16:5–33, 1995.

[3] P. J. Phillips, H. Wechsler, J. S. Huang, and P. J. Rauss. The FERET database and evaluation procedure for face-recognition algorithms. *Image and Vision Computing*, 16(5):295–306, 1998.

[4] N. A. Thacker, A. J. Lacey, and P. Courtney. An empirical design methology for the construction of computer vision systems. Technical report, Department of Imaging Science and Biomedical Engineering, Medical School, University of Manchester, UK, May 2003. `http://peipa.essex.ac.uk/benchmark/methodology/white-paper/methodology.pdf`.

[5] J. F. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.

[6] Paul R. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, 1995.

[7] Ron Kohavi and Foster J. Provost. Applications of data mining to electronic commerce. *Data Mining and Knowledge Discovery*, 5(1/2):5–10, 2001.

[8] S. M. Lucas and K. Sarampalis. Automatic evaluation of algorithms over the internet. In *Proceedings of the International Conference on Pattern Recognition*, volume 2, pages 471–474, 2000.

[9] Adrian F. Clark. HATE: A harness for algorithm testing and exploration. in preparation.

[10] Gnuplot. `http://www.gnuplot.info`.

[11] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer-Verlag, fourth edition edition, 2002.

[12] I. Aleksander, W. V Thomas, and P. A. Bowden. WISARD: a radical step forward in image recognition. *Sensor Review*, 4(3):120–124, 1984.