Experiment 7: Evaluation and Machine Learning

Adrian F. Clark

This experiment has two aims. Firstly, it will make you familiar with the mechanics of performance evaluation as currently practised by computer vision researchers and developers. The statistical comparisons you perform put you right at the forefront of what the research community is doing. Secondly, the algorithms you are comparing all employ machine learning, so this gives you a nice introduction to some of the techniques that are discussed in the final two lectures of the module.

Contents

1 Introduction

Machine learning is now ubiquitous in computer vision: there are very few real-world applications that do not make use of it. In fact, computer vision research has always been closely associated with machine learning because humans clearly learn to see.

The standard way of evaluating computer vision algorithms is statistical, generally employing a large number of ground-truthed images: images for which one or (better) more domain experts agree on what they contain. As most effective machine learning techniques also require ground-truthed imagery for training, the twin requirements of training and testing are closely related. In this experiment, you will train up a number of vision techniques and evaluate the trained systems on the popular MNIST example, which comprises 60,000 training images and 10,000 test ones. MNIST is now regarded as being quite easy, so some researchers have produced a drop-in replacement that uses images of different types of clothing; this fashion MNIST database is regarded as being somewhat harder. (In fact, I regard one of its images as being truly impossible.)

To carry out this experiment, you will need a zip-file of the software *etc*. This will not allow you to carry out all the training yourself as some of it will take too long on the machines in CSEE's Software Labs. However, if you want to do that on your own computer, perhaps because your project involves machine learning, there is a zip-file which includes all the training images ($\sim$ 300 Mbytes in size). All of the files in the smaller zip-file are contained in the larger one, so you won't need to download both.

2 Training and testing algorithms

The zip-file contains a program called `ml` (for "machine learning") which is able to train a system on a variety of vision tasks and test them, generating output that can be fed into FACT. Many of the machine learning algorithms discussed in lectures are implemented in it. The simplest way of using it is something like:

```
./ml -learner=svm train recog.kb train/*
```

This tells the program to `train`, saving the trained recognizer in file `recog.kb` (the "`.kb`" is for "knowledge base" but you can use any extension you like) and the remainder of the command-line arguments are the data files on which the recognizer is trained. The `-learner` qualifier tells `ml` to use a support vector machine for training; the possibilities are:

cnn: Convolutional Neural Network

eigen: The "eigenfaces" algorithm described in Chapter 8 of the notes. My implementation of this is particularly simplistic, which means it yields overly poor results and takes ages to run: $O(N^2)$ rather than $O(N)$.

mlp: Multi-Layer Perceptron

rf: Random Forests — not described in the lecture notes but quite widely used in practice

svm: Support Vector Machine

wisard: WISARD

so you can see I'm training a SVM here.

This approach of specifying all the training images on the command line is quite elegant but ultimately runs out of steam: the buffer into which the Unix shell expands wildcards ("`train/*`" above) is of finite size and if the number of filenames is very large, it can overflow — and this is certainly the case for the 60,000 training images in MNIST, which we shall look at here. As an alternative, you can specify a *task file* which details both the training images (and corresponding classes) and the subsequent tests. The zip-file contains one for all of MNIST, called `mnist.task`, and also one for a subset of it, `mnist-part.task`. As usual, these are text files so feel free to look at them. With a task file, the command would instead be something like:

```
./ml -learner=svm train recog.kb mnist.task
```

As well as training up recognizers, `ml` can test them. A typical invocation would then be:

```
./ml test recog.kb mnist.task
```

The recognizer saved from the training phase, `recog.kb` here, has the training algorithms used stored within it so that `ml` uses the appropriate algorithm when working through the tests. The output from `ml` is a FACT-compatible transcript which you can save into a file using command-line redirection in precisely the same way as in earlier experiments, and you use FACT on it in the same way.

Each of the machine learning algorithms in `ml` is configured to work fairly well on the MNIST task but where an algorithm has tuning parameters that affect its performance, they can be set using command-line qualifiers. For the WISARD algorithm for example, you can set both the number of image locations that form one of its "tuples" and the number of tuples via `ml`'s `-nlocs` and `-ntuples` qualifiers respectively.

If your project involves machine learning, you are welcome to use `ml` as the basis of your own software — but if you do, remember that you need to acknowledge it to avoid being accused of cheating. The ability to read task files and output FACT transcripts is especially useful.

learner	train time	test time
EIGEN	9	4,524
MLP	278	1
RF	68	2
SVM	250	88
WISARD	51	9

Figure 1: Training and testing times for different learners

3 Assessing algorithms' performance individually

The zip-file contains trained versions of the recognizers listed above on MNIST, in filenames such as `mnist-svm.kb` for SVM. I have also ran `ml` on them in `test` mode, yielding the transcripts stored in files `mnist-svm.res` and so on. The training and testing times in seconds on the author's (rather fast) laptop are shown in Figure 1. WISARD was run with `-ntup=50 -nlocs=10`. For all the learners but `SVM` which is deterministic, the random number generator was initialized with `-seed=1` on the `ml` command line. There is no result from CNN because, at the time of writing, Tensorflow was unhappy on the author's computer — and the CNN code in `ml` is untested for the same reason, though the code in the lecture notes was tested.

Your task is to analyse the performance of the various learning algorithms on these results using FACT — you should know how to do that from previous experiments. Try rank-ordering the algorithms in terms of accuracy. Would this order be the same if you ranked them in terms of specificity or some other measure?

If you have oodles of CPU time to spare, you might train up (say) the MLP recognizer a few times without fixing the seed on the command line, which means it will use a different series of random numbers in each run. Then use FACT to ascertain whether the different trained versions yield different accuracies. The commands involved are:

```
./ml train -learner=mlp -data=mnist/train haveago.kb mnist.task
./ml test -data=mnist/test haveago.kb mnist.task
```

You can also ascertain whether there are any statistically-significant differences in performance (see below) from the runs.

4 Comparing the performances of algorithms

As discussed in lectures, a good test harness should allow one to compare the performances of algorithms in a statistically-valid way. FACT does this by using McNemar's test and to use it, you run it in `compare` mode with a pair of transcript files — you should have done this in the experiment on content-based image retrieval:

```
./fact compare mnist-svm.res mnist-mlp.res
```

In fact, you can use the comamnd:

```
./fact compare mnist-*.res
```

in which case FACT does pair-wise comparisons between all the transcript files.

There are some subtleties when comparing more then two algorithms because you need to adapt the critical value for significance of 1.96 described in Chapter 6 of the lecture notes. When you choose a pair of algorithms to compare, you are choosing from an ensemble (to use the correct statistical nomenclature) of all possible algorithms. If you keep doing this many times, you will eventually choose a pair of algorithms for which there appears to be a significant difference in performance just because of the arrangement of the data. Remember, the critical value of 1.96 given in the lecture notes corresponds to an expectation that the data will make one algorithm appear to be better than another simply as a consequence of the data used one time on twenty — so if you perform twenty pairwise comparisons, one of them might be expected to appear significant simply because of the data and not because of a genuine performance difference. (Confusing, isn't it? Do talk to a demonstrator about this.)

What this means is that we need to *increase* the critical value that indicates significance so that a larger Z is needed from McNemar's test. The most widely-used such correction is the Bonferroni correction and comes down to multiplying the critical value by the number of algorithms being tested. You need to do this when interpreting the result from `fact compare`. When you have done this, you are in a position to judge which is the best algorithm to use on MNIST from all those considered in this experiment.

5 Training and testing a face recognizer

Having worked through the MNIST transcripts, it is time to do some training and testing yourself. You have been provided with a face database, the Olivetti Research Laboratory's one in the directory `orl` in the zip-file. This contains ten images of each of forty subjects. We shall retain 4 images of each subject for testing and use either five or six for training — see the files `orl5.task` and `orl6.task`.

Train up and test each of the learners on `orl5.task` and `orl6.task`, retaining their results. For example, to train and test an MLP on the ORL5 case, you'd use a command such as:

```
./ml train -learner=mlp -data=orl orl5-svm.kb orl5.task
./ml test -data=orl orl5-svm.kb orl5.task > orl5-svm.res
```

Then analyse and compare the results using FACT. Does using a larger number of training images make much difference to any of the learners? Are the any significant performance differences between learners with the same number of training images? What is your opinion of the experiment? Do discuss your answers with a demonstrator.