Experiment 2: Content-Based Image Retrieval

Adrian F. Clark

This experiment explores developing and assessing a complete computer vision application. The approach used identifies 'similar' images as being those that have similar histograms as determined by cross-correlation.

Contents

1 Introduction **2**

2 The strawman CBIR software **2**

3 Refining the CBIR software 3

4 Segmenting out the object 3

1 Introduction

Having devised a way of computing histograms as a way of describing an image, a natural thing to wonder is whether similar histograms mean that the corresponding images are similar. Perhaps surprisingly given how simple the idea is, this approach was used commercially in the early 1990s for precisely this reason. This type of technique is called *content-based image retrieval* (CBIR).

Rather than just run a single CBIR program on a dataset and obtain a few numbers, we shall go further: you will be given a 'benchmark' or 'strawman' program and try to improve it, then compare whether the improvement actually leads to better performance. This is close to what happens both in a research project and in product development.

2 The strawman CBIR software

The `cbir` program in the experiment's zip-file calculates the histogram of each image (i.e., there is one histogram even if the image is a colour one). This is discussed in detail in Chapter 3 of the lecture notes — you might like to read the relevant part before progressing through this experiment. We expect that `cbir` will be a poor approach because it does not distinguish colours, so an image with many bright blue pixels can be confused with one with (say) many bright green ones.

As well as the `cbir` program, the experiment's zip-file contains a set of images and a *test harness* called FACT, which runs a series of tests and keeps track of the number of successes and failures. Your first step is to assess the performance of the `cbir` program. First, ensure it and FACT are executable:

```
chmod +x cbir fact
```

and then run the test harness as described below.

The file `fruit.fact` contains the tests that are to be executed; it is human-readable and you are welcome to look at it. You should be able to run this test script on `cbir` using the command

```
./fact --interface=cbir-if execute fruit
```

This command tells FACT to run each test in `fruit.fact` on `cbir` and output a "transcript" of what happens; you can use `run` rather than `execute` if you prefer. The `--interface` part of the command gives it some Python code that invokes your `cbir` program for each test. When you run the above command, FACT will write output to your terminal window. The first line contains some identification information, used for checking in the analysis stages, and it is followed by one line per test. These lines are actually the transcript. To save the transcript in a file you simply use command-line redirection:

```
./fact --interface=cbir-if execute fruit > cbir.res
```

and twiddle your thumbs while it runs. If you don't want to wait, I've provided `cbir.res` in the experiment's zip-file.

The reason that `cbir` runs so slowly is that the histogram routine iterates over all the pixels itself. If you want to speed the program up, you can replace the call to your histogram routine with an invocation of the OpenCV or numpy histogram routine; or write you can out the individual histograms to files on the first iteration and read them back in on subsequent ones — or, with this number of images, simply cache them in a Python dictionary.

Having generated the transcript, the next stage is to analyze it. This is both quick and easy:

```
./fact analyse cbir.res
```

At this juncture, you are most interested in the accuracy. You can have FACT output more detail by appending `--detail=2` to the command.

3 Refining the CBIR software

As discussed above, the histogram that `cbir` calculates is not really sophisticated enough to perform the task effectively. Your next step is improve the algorithm and ascertain whether it actually delivers better performance.

Make a copy of `cbir` into a program called `cbir3` (for three histograms) and make the latter

- calculate separate histograms for red, green and blue;
- join together these three histograms end-to-end into one long one with $3 \times 256 = 768$ bins

as illustrated in Figure 1. Rather than look for a clever OpenCV routine to do the joining, you'll probably find it easier to write Python code to do it.

Then go through the same stages as before but with the interface file I've provided that tells FACT to run your `cbir3`:

```
chmod +x cbir3
./fact --interface=cbir3-if execute fruit > cbir3.res
./fact analyse cbir3.res
```

You can do an additional step, comparing the transcripts of your `cbir3` with the one from `cbir`:

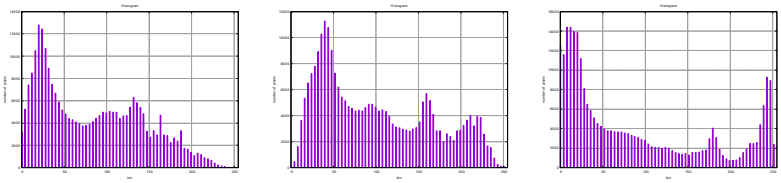
```
./fact compare cbir.res cbir3.res
```

Again, you can get more detail by appending `--detail=2` to the command.

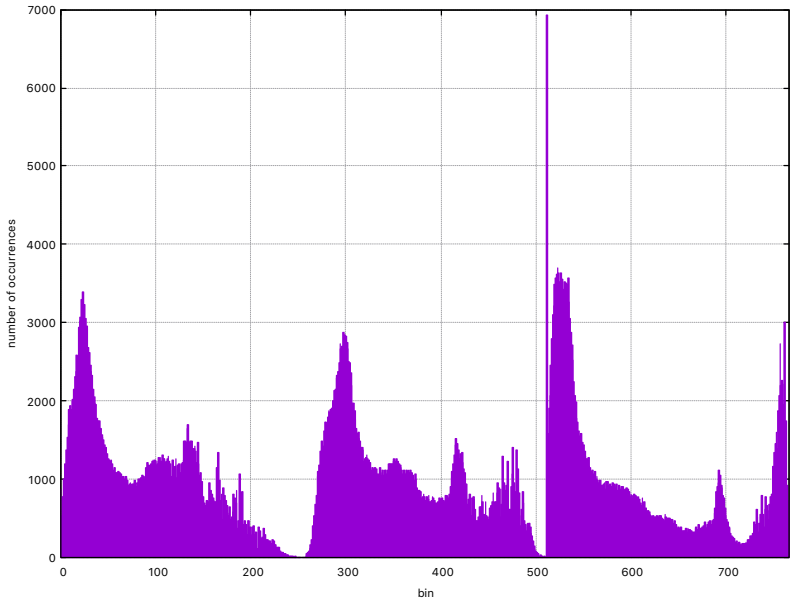
4 Segmenting out the object

Although all the images were captured on the same background, the fact that the fruit differ in size means that the number of background

(a) Image



(b) Separate histograms for red, green and blue channels



(c) Individual channel histograms joined end-to-end into one long one

Figure 1: Joining red, green and blue histograms

pixels will vary. It might be possible to overcome this by segmenting out the object from the background, and this can be done in the following way.

Use `xv` to look at several of the images. When you press the middle mouse button, it displays on the image the RGB and HSV values of the pixel under the cursor. Make a note of the range of hue values you encounter in the background.

Then make a copy of your `cbir3` in `cbir3s` ("`cbir` segmented"). In the latter, convert the image to HSV and use the hue limits you have identified to identify background pixels; do check that the HSV values of a pixel from `xv` are consistent with those computed OpenCV, which you can do my printing out (say) the HSV values of the top-left pixel of the image. Then make `cbir3s` convert all of these background pixels to white.

Calculate the histograms of these images with the whitened background as in your `cbir3`; but when you combine them into a long one, omit the very highest one (which corresponds to white). Then run it through FACT in the same way as before

```
./fact --interface=cbir3s-if execute fruit > cbir3s.res
```

and compare the transcripts from `cbir3` with `cbir3s` (and with the original `cbir`).

Finally, you should reflect on the improvements made to `cbir` to yield `cbir3` and `cbir3s`. What was each of the improvements intended to achieve? Did it make a difference? Has the testing been thorough enough? How could it be improved? We shall consider these aspects of experimental design when we look at evaluating vision systems later in the module.