CS 3430: SciComp with Py
Assignment 9
Direct Correlation and Image Histograms

Vladimir Kulyukin
Department of Computer Science
Utah State University

February 25, 2017

1 Learning Objectives

1. Direct Correlation in DPIV
2. Image Histograms
3. Image Indexing and Retrieval

Introduction

In this assignment, we'll implement the direct correlation method used in DPIV and test it on a few matrices. I decided not to include in this assignment the expriments with OpenPIV. If you're interested in exploring DPIV on your own, watch the last 10 minutes of the screencast CS3430_S20_Lec19_Parts_01_02.mp4 on Canvas (Lecture 19) on how to install OpenPIV and use it to to generate vector fields from flow images. We'll also implement a simple image indexing and retrieval engine. You'll save your solutions to problem 1 in `dcorr.py` and to problem 2 in `hinx.py` and `hret.py` and submit these three files in Canvas. The file `cs3430_s20_hw09_uts.py` contains my unit tests for this assignment.

Problem 1 (2 points)

In the file `dcorr.py`, implement the function `direct_corr(fixed, dancer)` that takes two 2D numpy matrices, `fixed` and `dancer`, and returns a 2D numpy array `C` with their correlation coefficients, as discussed in the first 35 minutes in the screencast CS3430_S20_Lec19_Parts_01_02.mp4 for Lecture 19 and the pdf slides for this part of the lecture. This file contains the method `__comp_mats(self, m1, m2, err=0.0001)`.

```
def __comp_mats(self, m1, m2, err=0.0001):
    if m1.shape != m2.shape:
        return False
    nr, nc = m1.shape
    for r in range(nr):
        for c in range(nc):
            if abs(m1[r,c] - m2[r,c]) > err:
                return False
    return True
```

This method returns `True` if the corresponding cells in two numpy matrices, `m1` and `m2`, are the same at a given error level. Feel free to play with different error levels. This method is used in the first 10 unit tests I wrote for this problem. All test have the same structure. So, let's take a quick look at one. I removed the print statements from the definition for brevity.

```
def test_hw09_prob01_ut01(self):
    M1 = np.array([[10, 0, 0],
                   [0, 0, 0],
                   [0, 0, 0]],
                  dtype=float)
    M2 = np.array([[0,  0, 0],
                  [0, 10, 0],
                  [0,  0, 0]],
                 dtype=float)
        M12 = np.array([[0,   0,  0, 0, 0],
                        [0, 100,  0, 0, 0],
                        [0,   0,  0, 0, 0],
                        [0,   0,  0, 0, 0],
                        [0,   0,  0, 0, 0]],
                       dtype=float)
        C12 = direct_corr(M1, M2)
        assert self.__comp_mats(C12, M12)
        C21 = direct_corr(M2, M1)
        M21 = np.array([[0,  0,  0,   0, 0],
                        [0,  0,  0,   0, 0],
                        [0,  0,  0,   0, 0],
                        [0,  0,  0, 100, 0],
                        [0,  0,  0,   0, 0]],
                       dtype=float)
        assert self.__comp_mats(C21, M21)
```

The unit test defines two matrices, `M1` and `M2`. The matrix `M12` is the ground truth direct correlation matrix of direct correlation b/w `M1` and `M2` with the former being the FIXED matrix and the latter – the DANCER matrix. The line `C12 = direct_corr(M1, M2)` computes the direct correlation matrix `C12`. The next line compares `M12` and `C12` with `self.__comp_mats(C12, M12)`. The matrix `M21` is the ground truth matrix of computing direct correlation b/w `M1` and `M2` with `M2` being he FIXED matrix and `M1` – the DANCER matrix. The variable names in the other 9 tests have the same semantics.

Problem 2 (3 points)

An image retrieval task can be formulated as follows. Given a collection of images, index each image in the collection in terms of some features present in the image. This step is called *indexing*. The output of the indexing step is a collection index. This index can be an SQL database, a text file, or a persisted object such as a Py dictionary. Given an input image and a collection index generated in step 1, find and display the image most similar to the input image. This step is called *retrieval*. Of course, we can relax the similarity retrieval requirement and retrieve the top 10/20/100 similar images.

I uploaded the following zip archives under Canvas announcements for Assignment 09 for this problem. The archives are:

- `hist_img_01.zip`;
- `hist_img_02.zip`;
- `hist_img_03.zip`;
- `hist_img_04.zip`;
- `hist_img_05.zip`;
- `hist_img_06.zip`;
- `hist_img_07.zip`;
- `hist_test.zip`.

Download and unzip all images into the folder `images`. There should 318 images of Logan streets and some lunch tray images from a Logan school. The lunch tray images are courtesy of Dr. Heidi Wengreen, a USU nutrition professor, with whom I collaborated on a food texture recognition project several years ago.

We'll use the images in the directory `images` for persistent indexing (aka pickling). The archive `hist_test.zip` contains two directories: `car_test` and `food_test`. Unzip `ht_test.zip` into the same directory where you created the directory `images`. We'll use the test images for image retrieval. Don't mix them with the indexing images. As I mentioned in the screencast CS3430_S20_Lec19_Part03.mp4 for Lecture 19, it is possible to compute RGB, HSV, and grayscale histograms. We won't use grayscale histograms in this assignment.

Histograms are worth considering any time one needs to retrieve images similar to a given image from a database of images or finding an object in an image. Histogram matching is based on the implicit assumption that similar images have similar histograms. This, however, is not always true. For example, a blue ball will have a histogram similar to a blue cube and quite different from the histogram of a red ball. Therefore, histograms may not be sufficient without shape or some other geometrical information.

Indexing Images

Implement the function `hist_index_img` in `hinx.py`. This function takes an image path `imgp`, a string specification of the color space (this can be either `'rgb'` or `'hsv'`), and the number of bins in each color channel in the histogram (we'll use 8 and 16 in this assignment). Recall that the number of bins parameter is used in the OpenCV `cv2.calcHist()` function. For example, if you want to use 8 bins for each channel to index the image `img`, the call to `cv2.calcHist()` may look as follows:

```
cv2.calcHist([img], [0, 1, 2], None, [8, 8, 8], [0, 256, 0, 256, 0, 256])
```

The function `hist_index_img` places the key-value pair `(imgp, norm_hist)` in the global dictionary `HIST_INDEX`. Depending on the value of `color_space`, the `norm_hist` is either the normalized and flattened RGB histogram of the image in `imgp` or the normalized and flattened HSV histogram of the same image. You can watch CS3430_S20_Lec19_Part03.mp4 for Lecture 19 on how to normalize and flatten histograms. The files `histo_03.py` and `histo_04.py` that I posted in the materials for Lecture 19 give you examples on how to normalize and flatten histograms. Let's run a unit test.

```
    def test_hw09_prob02_ut01(self):
        HIST_INDEX = {}
        hist_index_img_dir(Assign09UnitTests.IMGDIR, 'rgb',
                           8,
                           Assign09UnitTests.PICDIR + 'rgb_hist8.pck')
```

This test generates the following output:

```
../images/
indexing ../images/16_07_02_14_23_48_orig.png
../images/16_07_02_14_23_48_orig.png indexed
indexing ../images/16_07_02_15_18_56_orig.png
...
indexing ../images/16_07_02_14_50_47_orig.png
../images/16_07_02_14_50_47_orig.png indexed
indexing finished
```

Running this test creates a pickle file `rgb_hist8.pck` inside `Assign09UnitTests.PICDIR` (change the value of this variable in the the unit test file as you see fit) that contains the persisted version of `HIST_INDEX`. Unit tests 2, 3, and 4 for Problem 2 create and persist three more dictionaries. You don't have to write any code for pickling dictionaries. The function `hist_index_img_dir()` is defined for you.

Retrieving Images

After we've created the persisted dictionaries (aka indexes), we can load them into Python with the function `load_hinx()` defined in `hret.py`.

```
def load_hinx(pick_path):
  with open(pick_path, 'rb') as histfile:
    return pickle.load(histfile)

>>> hist_index = load_hinx('rgb_hist8.pck')
>>> len(hist_index)
>>> 318
```

Unit tests 5 – 8 use `load_hinx()` to check that each persisted dictionary contains 318 items.

Implement the function `find_sim_rgb_images(imgpath, num_bins, hist_index, hist_sim)` in `hret.py` that finds and displays the top 3 images most similar to the image in `imgpath` by computing the similarity scores between this image and all images in a given histogram dictionary `hist_index`. The parameter `num_bins` should be 8 or 16. Use the following strings as values of `hist_sim` (i.e., the histogram similarity metric): `correl` – for correlation, `chisqr` – for chi square, `inter` – for intersection, and `bhatta` – for bhattacharrya. Watch the screencast CS3430_S20_Lec19_Part03.mp4 for Lecture 19 to learn more about these metrics. The top three images and the input image are displayed in four separate matplotlib figures (see the function `show_images()`) along with their similarity scores. The similarity scores are printed out. This function computes a list of 2-tuples of the form `(path, sim_score)`, where `sim_score` is the similarity score computed with `hist_sim` between the image in `imgpath` and the image in `path`. This list of 2-tuples (i.e., the match list) is then sorted from lowest to highest or from highest to lowest similarity score, depending on the value of `hist_sim`, because for some similarity metrics the higher the better, for others – vice versa. The list of the top 3 matches is returned.

Let's do a test where we use the 8-bin rgb index to find the top 3 similar images by using the histogram intersection similarity function (i.e., `cv2.HISTCMP_INTERSECT`).

```
    def test_hw09_prob02_ut09(self):
        hist_index = load_hinx(Assign09UnitTests.PICDIR + 'rgb_hist8.pck')
        assert len(hist_index) == 318
        imgpath = Assign09UnitTests.TSTDIR + 'food_test/img01.JPG'
        print(imgpath)
        top_matches = find_sim_rgb_images(imgpath,
                          8, hist_index, 'inter')
        for imagepath, sim in top_matches:
            print(imagepath + ' --> ' + str(sim))
        inimg = cv2.imread(imgpath)
        show_images(inimg, top_matches)
        del hist_index
```

Here is my output.

```
images/123461762.JPG --> 2.6907286450397976
images/123465049.JPG --> 2.6331934205607297
images/123472255.JPG --> 2.435314836443297
```

The function `show_images(imgpath, match_list)` in `hret.py` takes the path to the image you're matching and the match list returned by `find_sim_rgb_images()` or `find_sim_hsv_images()` (see below) and displays the image and the top 3 matching images. Uncomment calls to it if you want to display the images as you run the unit tests.

Implement the function `find_sim_hsv_images(imgpath, bin_size, hist_index, hist_sim)` that behaves in the same way as `find_sim_rgb_images()`, except `hist_index` is an HSV dictionary. The unit tests for Problem 2 contains my top three matching images for each retrieval. In my opinion, the quality of retrieved images is much better in RGB than in HSV for this assignment.

What To Submit

Submit `dcorr.py`, `hinx.py`, and `hret.py` in Canvas. Don't submit any images or pickled dictionaries. And remember to work on one unit test at a time.

Happy Hacking!