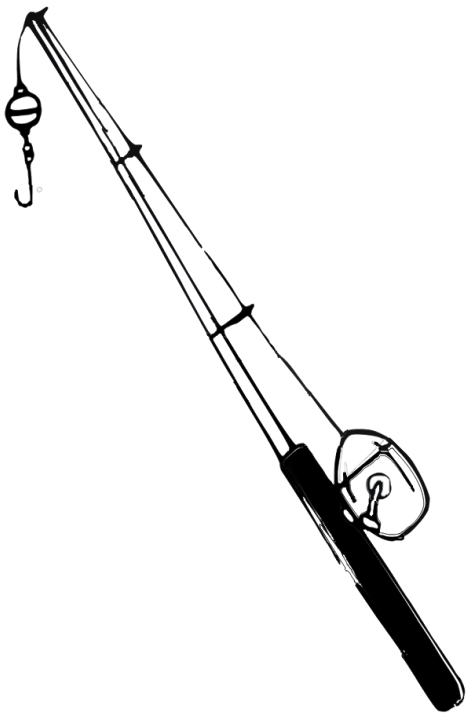
VISUAL Information Retrieval



COMS W4735: Visual Interfaces to Computers

Assignment 2
due: 3/17/2015
Nina Baculinao
uni: nb2406

Introduction

The goal of the Visual Information Retrieval system is to explore different ways of deciding the degree of similarities between actual images. It primarily looks at color, texture, similarity as a linear sum of ratios between the two, and clustering. Finally, the system is evaluated against the more intelligent baseline of human judgment to see how it performs.

Images analyzed in this project had the following properties:

- PPM format (versus lossier JPEG versions)
- 89 x 60 pixels
- 01-40 numbering sequence

Hardware and library specifications of this project are as follows:

- MacBook Air 11 inch running on OSX Yosemite
- Python Standard Library 2.7.9
- OpenCV with a Python binding, v2.4.10.1
- NumPy 1.9.1
- Matplotlib 1.4.2

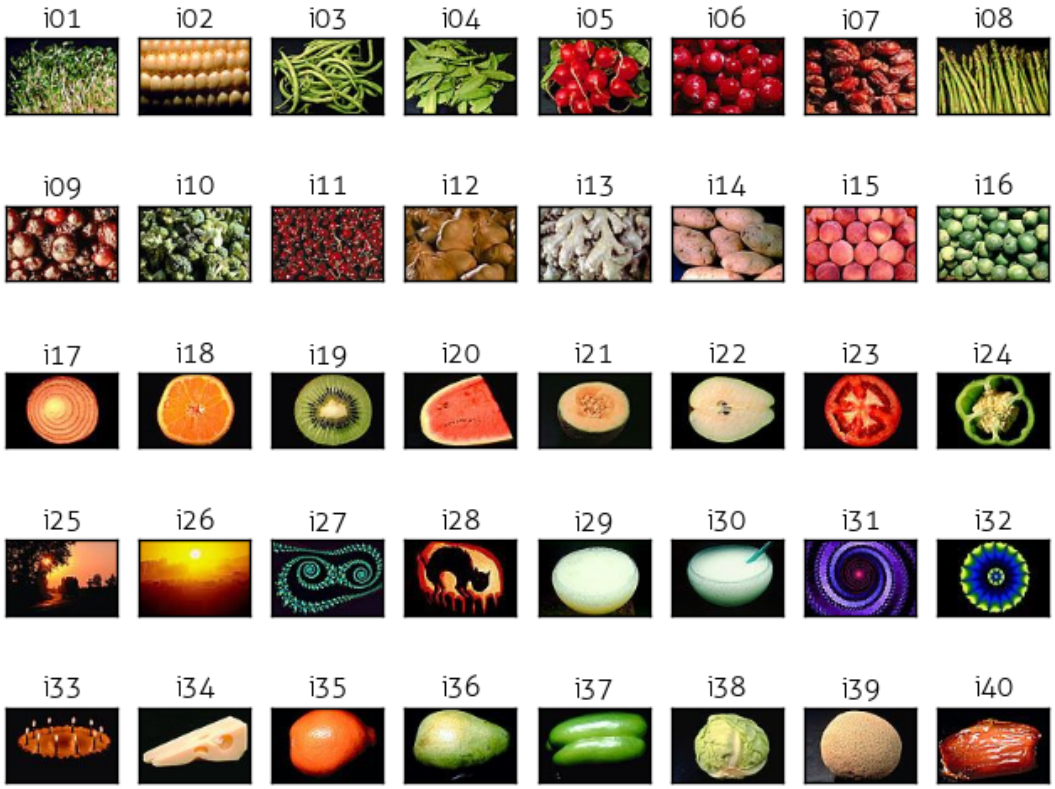


Figure 1. All the images analyzed by the system

Note: the display on the previous page was generated with matplotlib using the following code snippet. While the learning curve for matplotlib is a bit steep, its ability to plot images and graph figures along various axes, rows and columns was very useful.

```
def display_all(images, titles):
    plt.rcParams['font.family']='Aller Light'
    for i in xrange(NUM_IM):
        plt.subplot(5,8,i+1),plt.imshow(cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB))
        plt.title(titles[i], size=12)
        plt.xticks([]),plt.yticks([])
    title = './all_im.png'
    plt.savefig(title, bbox_inches='tight')
    print title
```

Gross Color Matching

Overview

The algorithm to compare images strictly on their total color distribution is described in broad steps directly below, and in more detail following this general description.

1. For each image in the directory:
    a. Load the image with OpenCV2 (and append to a list)
    b. Generate a color histogram for the image (and append to a list)
    c. *Optional*: Visualize the histogram as a bar graph (and append to a list)
2. Calculate the distance between every image pair (and store in a lookup dictionary)
    a. Use the histograms values of each image pair to determine the L1 norm
3. For each image:
    a. Determine the three images *most like it* and the three images *most unlike it* in terms of color distribution
    b. Display the septuples: original, three alike, three unalike
4. Determine group of four *most alike* images and group of four *most unalike* images
    a. Display the two groups of four: most alike, most unalike

Bin Size

The chosen bin size and black threshold for my program are as follows:

```
COL_RANGE = 256
BLK_THRESH = 40
BINS = 8
BIN_SIZE = int(COL_RANGE/BINS)
BLK_THRESH = 40
```

Because there are 256 RGB possible values in each color pixel, giving each color value a separate bin would make generating a histogram unfeasible because that would be accounting for ~16 million bins! Not to mention, there would certainly be a scarcity of data for each bin. Therefore, it was critical to determine a good size for clustering color values that would still capture the color distribution of an image histogram within decent runtime. As can be expected, I was inclined toward testing values to the power of 2. While it is true that the color histograms generated with 16 and 32 bins had a wider and more diverse array of color bars compared to the histogram visuals (on the following page), I found out that I got very comparable results for best matching images and final septuple of color matching, whether I chose 8, 16 or 32 bins. Therefore I decided 8 was good enough, and fast enough. There were still 8x8x8 = 512 possible bins for the RGB values to go!

Black Threshold

The black threshold was another important decision to be made, as several images had black backgrounds, and could results in false negative color matches if the histogram matched similar black background content of say, an orange, with a kiwi.

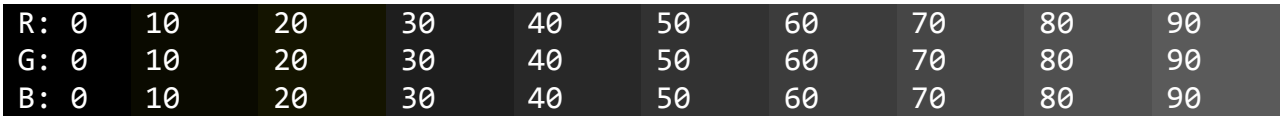


Figure 2. Color bar showing range of RGB values. 40,40,40 still looks black, right?

After printing out the numpy array and surveying the black, I determined with my human eyes as a baseline that values under 40, 40, 40 are hard to discern as other than black. Is it just me? A possible influence for my low sensitivity to black color may be due to the fact that I like to do computer work with flux on and low brightness.

Therefore, after laying down the law with a black threshold of 40, I created a condition in my color_histogram function to bar all black and near-black pixels from taking a coveted spot in my histogram. The if condition is as follows, where I check the pixel before adding to histogram whether *all* of its RGB values are over 40:

```
if pixel[0] > BLK_THRESH and pixel[1] > BLK_THRESH and pixel[2] > BLK_THRESH:
```

Color Histogram

The histogram calculations are computed without the aid of any black box algorithms. While OpenCV does have the cv2.compareHist function and SciPy also comes with its own distance metrics, implementing a 3D color histogram is actually quite simple, as can be seen in my code fragment below:

```
def color_histogram(image, title):
    '''
    Calculate the 3D color histogram of an image by counting the number
    of RGB values in a set number of bins
    image -- pre-loaded image using cv2.imread function
    title -- image title
    (optional: visualize the histogram as a bar graph)
    '''
    colors = []
    h = len(image)
    w = len(image[0])
    # Create a 3D array - if BINS is 8, there are 8^3 = 512 total bins
    hist = np.zeros(shape=(BINS, BINS, BINS))
    # Traverse each pixel in the image matrix and increment the appropriate
    # hist[r_bin][g_bin][b_bin] - we know which one by floor dividing the
    # original RGB values / BIN_SIZE
    for i in xrange(h):
```

```
        for j in xrange(w):
            pixel = image[i][j]
            # If the pixel is below the black threshold, do not count it
            # Also note: pixel[i] is descending since OpenCV loads BGR
            if pixel[0] > BLK_THRESH and pixel[1] > BLK_THRESH \
            and pixel[2] > BLK_THRESH:
                r_bin = pixel[2] / BIN_SIZE
                g_bin = pixel[1] / BIN_SIZE
                b_bin = pixel[0] / BIN_SIZE
                hist[r_bin][g_bin][b_bin] += 1
                # Generate list of color keys for visualization
                if (r_bin,g_bin,b_bin) not in colors:
                    colors.append( (r_bin,g_bin,b_bin) )
    # plot = visualize_chist(image, hist, colors, title)
    # return plot, hist
    return hist
```

Important decisions in this step included selection of the **BINS** and **BLK_THRESH**, which I already discussed.

You might notice that the invocation for the **visualize_chist** function is commented out. That is because writing and generating every single figure takes about a little bit of extra runtime, and once I saved the images to a folder, I do not really need to produce the bar graphs again again. The **visualize_chist** function invokes matplotlib to plot a bar graph for the histogram, so it's a lot of boilerplate, but the key to it lies in this line:

```
colors = sorted(colors, key=lambda c:
-hist[(c[0])][(c[1])][(c[2])])
```

colors is the list of RGB color tuples from the **color_histogram** method, and this line re-sorts it from the highest count for that tuple in the histogram to the lowest count. Then it uses the following helper function:

```
def hexencode(rgb, factor):
    """Convert RGB tuple to hexadecimal color
code."""
    r = rgb[0]*factor
    g = rgb[1]*factor
    b = rgb[2]*factor
    return '#%02x%02x%02x' % (r,g,b)
```

to create the color hex string for displaying the bar graph. Example output can be seen on the right.

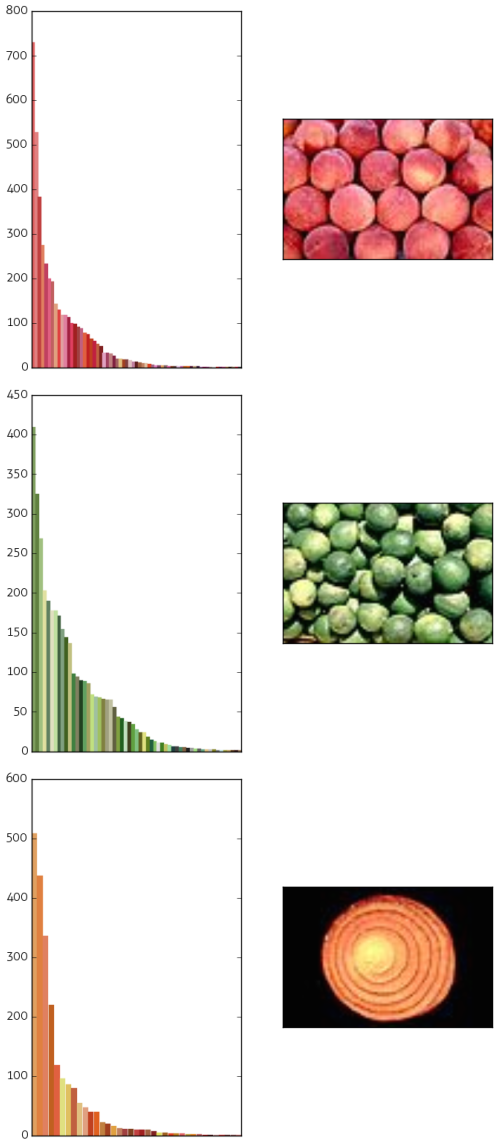


Figure 3. Histograms for images: i15, i16, and i17. Note that black pixels are not counted.

Color Similarity and Distance

To calculate the relative similarity or distances between a pair of images, I used the suggested L1_norm as discussed in class. In my particular implementation, I considered the norm (or distance) for two images to be:

$$L1\_norm = \sum (differences) / \sum (pixel\ count)$$

where

$$L1\_norm = distance = 1 - similarity$$

or you could say

$$similarity = 1 - distance = 1 - L1\_norm$$

Since it is very easy to convert between similarity and distance, and I found it more intuitive to work with distances than similarities, looking for images with "shorter distances" and closer to 0, I generally tended to use a list of distance values in my code. However, for the sake of human readability and because we often talked of similarity measures in class and in discussions of this assignment, I printed similarity values for the septuple captions for my color matches.

My simple implementation is below:

```
def l1_color_norm(h1, h2):
    diff = 0
    total = 0
    for r in xrange(0, BINS):
        for g in xrange(0, BINS):
            for b in range(0, BINS):
                diff += abs(h1[r][g][b] - h2[r][g][b])
                total += h1[r][g][b] + h2[r][g][b]
    l1_norm = diff / 2.0 / total
    similarity = 1 - l1_norm
    return l1_norm
```

For constant time lookup, I decided to calculate all possible pairwise distances between images and store them in a lookup table called chist_dis.

```
def calc_cdistance(chists):
    chist_dis = {}
    for i in xrange(NUM_IM):
        for j in xrange(i, NUM_IM):
            if (i,j) not in chist_dis:
                d = l1_color_norm(chists[i], chists[j])
                chist_dis[(i,j)] = d
    return chist_dis
```

Septuple of Three Best and Three Worst Matches

The existence of my lookup table for every pair made finding the three best and three worst color matches quite efficient. The function below takes in one image, and compares it to every other image in the image set, then sorts their distances in a list by smallest to greatest distance, returning the first three items (the best matches) and the last three items (the worst matches) in the list. Within my main method, I run color_matches for all 40 images to generate 40 septuple sets.

```
def color_matches(k, chist_dis):
    """
    Find images most like and unlike an image based on color distribution.
    k -- the original image for comparison
    chists -- the list of color histograms for analysis
    """
    results = {}
    indices = []
    distances = []

    for i in xrange(0, NUM_IM):
        if k > i: # because tuples always begin with lower index
            results[i] = chist_dis[(i,k)]
        else:
            results[i] = chist_dis[(k,i)]
    # Ordered list of tuples (dist, idx) from most to least similar
    # -- first value will be the original image with diff of 0
    results = sorted([(v, k) for (k, v) in results.items()])
    # print 'results for image', k, results
    seven = results[:4]
    seven.extend(results[-3:])
    distances, indices = zip(*seven)
    return indices, distances
```

Here is an example of my output. Full results can be found at the end of this section and the full list with color histograms + image septuples appears in the appendix appendix.

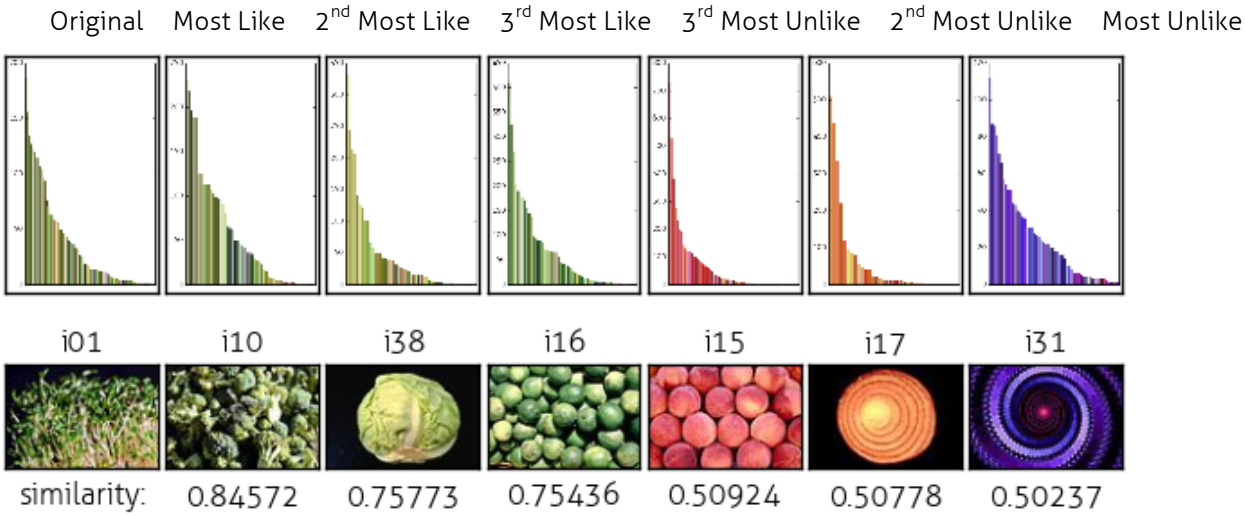


Figure 4. Septuple of three most alike images and three most unlike images to original

Group of Four "Most Alike" and "Most Unlike"

It was here that I had to adjust my original plan/algorithm because of low performance for selecting the "Four Most Unlike" images. Originally I just picked my group of four my collection septuples. I would look at the original image plus its three doppelgangers or the original image plus its three anti0images, and consider that a valid candidae for the "group of four." From these lists, I took the four most alike with lowest combined sum for distance, and four most unalike with highest combined sum for distance. The approach returned to me gave me was "matching set" for i01.ppm (good) and "unmatching set" for i03.ppm (not so good – as you can see in the first column).

This lazy and naïve strategy was fine for the most alike set, but backfired for the most different set because as you can see i06 and i15 are quite close to each other in terms of color and are in the same set because both oppose green beans. I ended up storing all combinations of four as a combination of distances between *every single pair* in the group, and then picking the group with the greatest distance. In the better results, there nothing matches i15. Results and implementation below.[1]

Figure 5: Not-So-Good: Alike, Unalike Better Results: Alike, Unalike


```
def find_four(chist_dis):
    results = {}
    # ensure that a<b, b<c and c<d as order does not matter
    for a in xrange(NUM_IM):
        for b in xrange(a+1,NUM_IM):
            for c in xrange(b+1,NUM_IM):
                for d in xrange(c+1,NUM_IM):
                    results[(a,b,c,d)] = \
                    chist_dis[(a,b)] + chist_dis[(a,c)] + \
                    chist_dis[(a,d)] + chist_dis[(b,c)] + \
                    chist_dis[(b,d)] + chist_dis[(c,d)]
    results = sorted([(v, k) for (k, v) in results.items()])
    best = results[0]
    worst = results[-1]
    indices = list(best[1])
    indices.extend(list(worst[1]))
return indices
```

[1] Note that counting all combinations of four elements created 91,390 entries. Though it could have been worse, as I considered order not to matter, and did not recalculate (a,b,c,d) and (d,c,b,a). Also, I did not have to recalculate the distances and could just use my lookup dictionary.

Full Results

Four Best Match:

Four Worst Match:

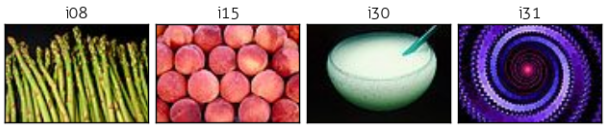
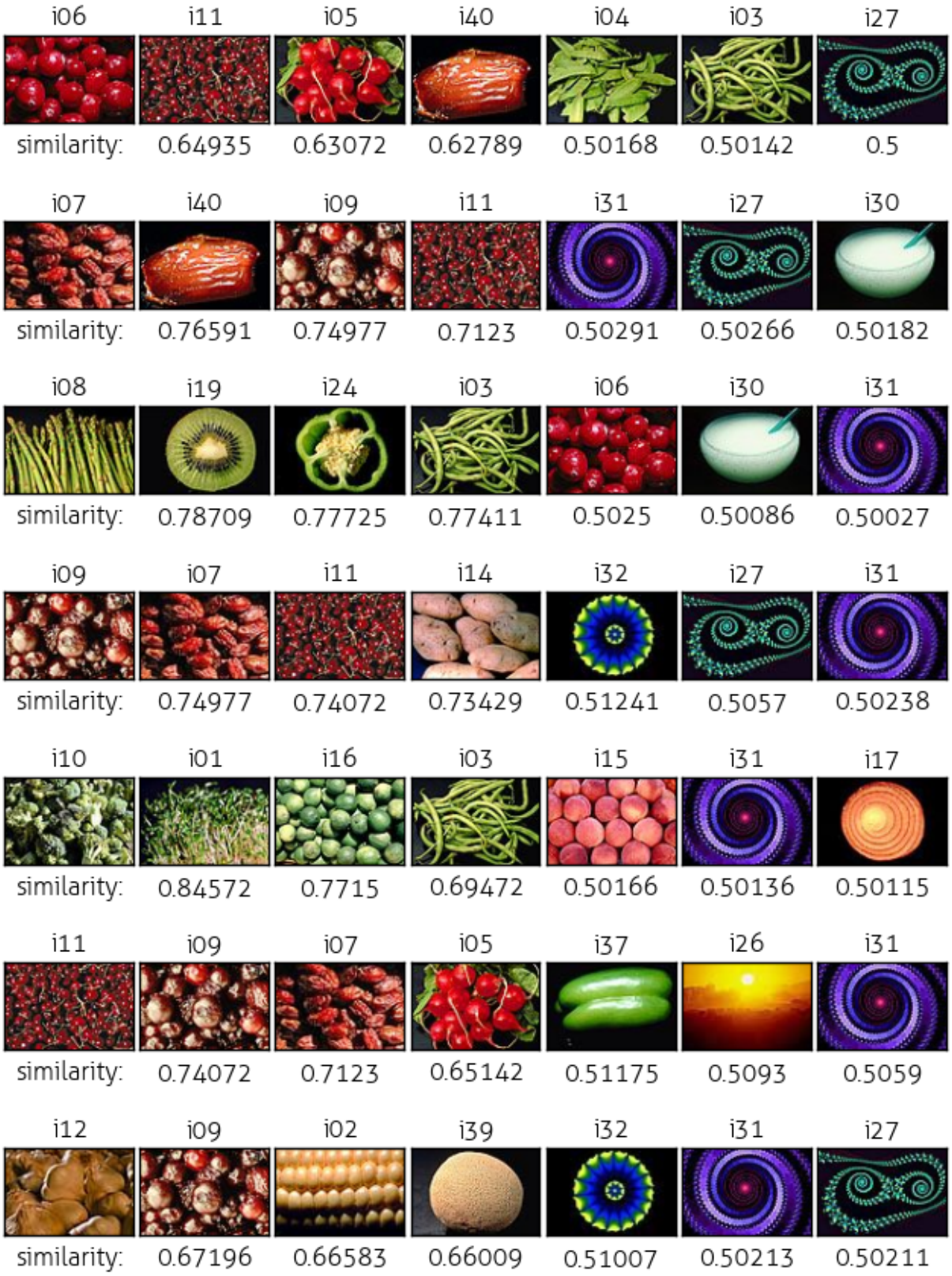
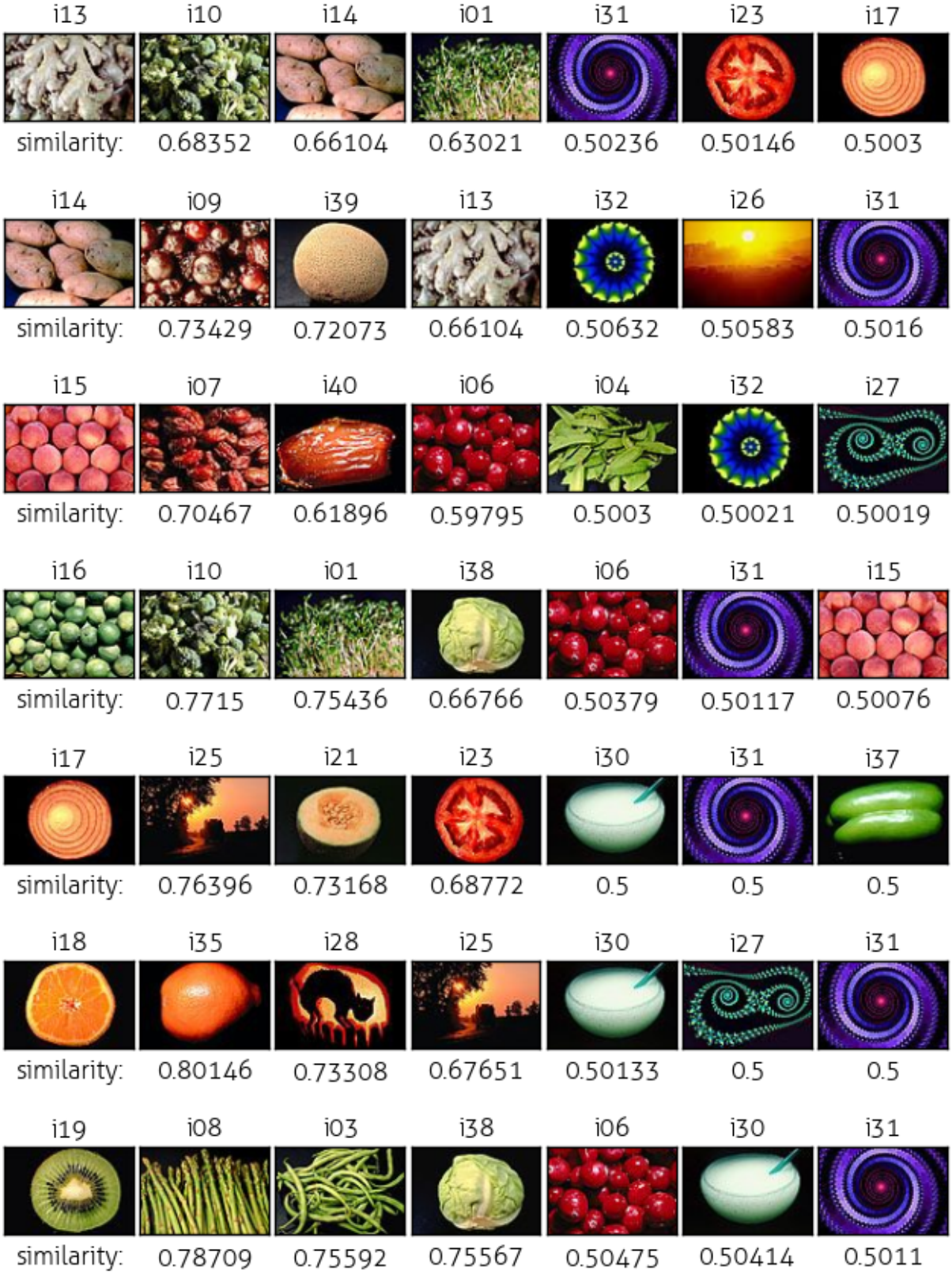
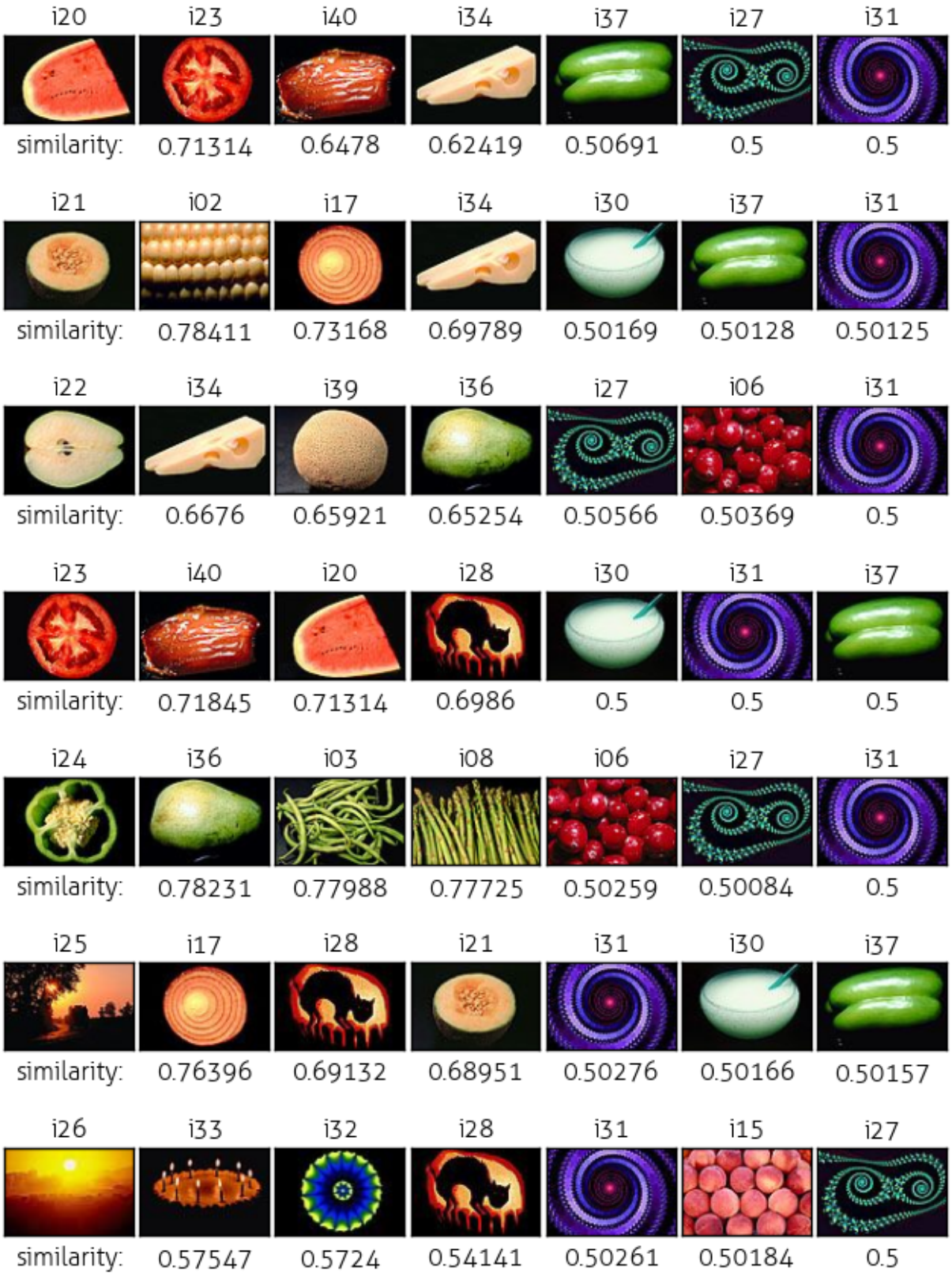


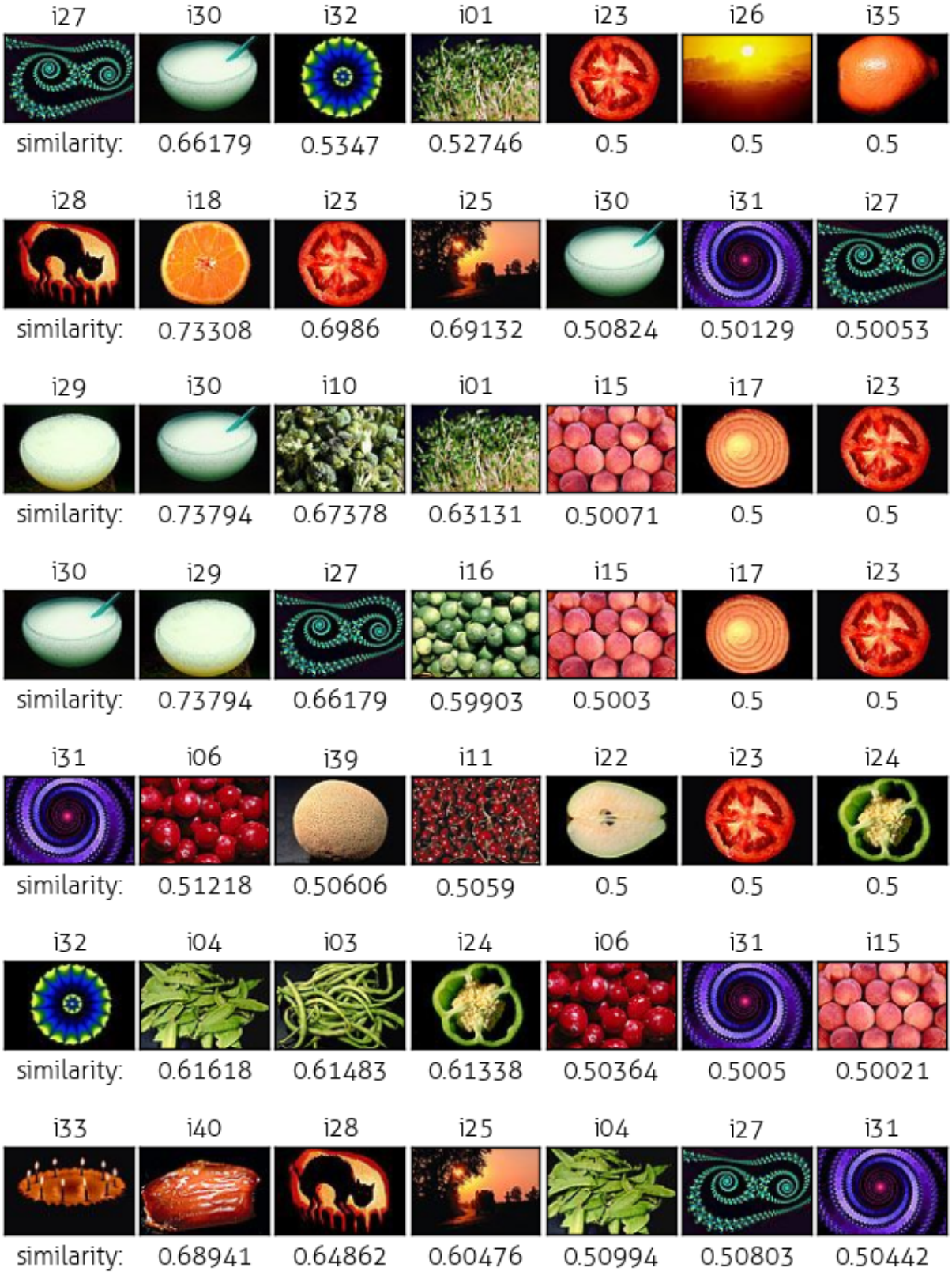
Image Septuples

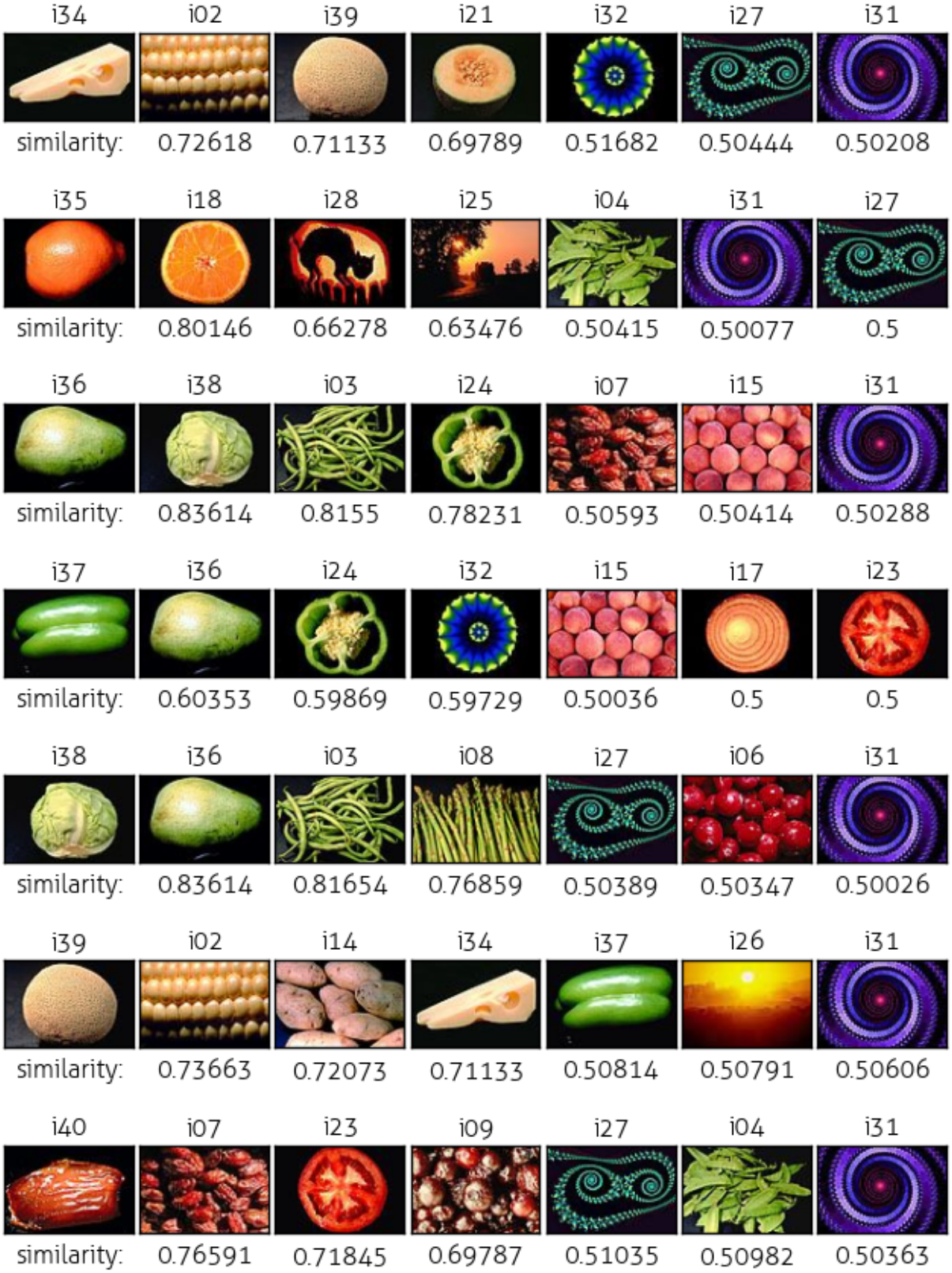
Original	Most Like	2nd Most Like	3rd Most Like	3rd Most Unlike	2nd Most Unlike	Most Unlike
i01	i10	i38	i16	i15	i17	i31
similarity:	0.84572	0.75773	0.75436	0.50924	0.50778	0.50237
i02	i21	i39	i34	i06	i27	i31
similarity:	0.78411	0.73663	0.72618	0.50639	0.50222	0.50156
i03	i04	i38	i36	i31	i06	i15
similarity:	0.85601	0.81654	0.8155	0.50162	0.50142	0.50031
i04	i03	i36	i24	i06	i31	i15
similarity:	0.85601	0.74297	0.74101	0.50168	0.50155	0.5003
i05	i03	i01	i24	i26	i30	i31
similarity:	0.66707	0.66349	0.65765	0.52189	0.51823	0.50485


i06 i11 i05 i40 i04 i03 i27
similarity: 0.64935 0.63072 0.62789 0.50168 0.50142 0.5
i07 i40 i09 i11 i31 i27 i30
similarity: 0.76591 0.74977 0.7123 0.50291 0.50266 0.50182
i08 i19 i24 i03 i06 i30 i31
similarity: 0.78709 0.77725 0.77411 0.5025 0.50086 0.50027
i09 i07 i11 i14 i32 i27 i31
similarity: 0.74977 0.74072 0.73429 0.51241 0.5057 0.50238
i10 i01 i16 i03 i15 i31 i17
similarity: 0.84572 0.7715 0.69472 0.50166 0.50136 0.50115
i11 i09 i07 i05 i37 i26 i31
similarity: 0.74072 0.7123 0.65142 0.51175 0.5093 0.5059
i12 i09 i02 i39 i32 i31 i27
similarity: 0.67196 0.66583 0.66009 0.51007 0.50213 0.50211


i13 i10 i14 i01 i31 i23 i17
similarity: 0.68352 0.66104 0.63021 0.50236 0.50146 0.5003
i14 i09 i39 i13 i32 i26 i31
similarity: 0.73429 0.72073 0.66104 0.50632 0.50583 0.5016
i15 i07 i40 i06 i04 i32 i27
similarity: 0.70467 0.61896 0.59795 0.5003 0.50021 0.50019
i16 i10 i01 i38 i06 i31 i15
similarity: 0.7715 0.75436 0.66766 0.50379 0.50117 0.50076
i17 i25 i21 i23 i30 i31 i37
similarity: 0.76396 0.73168 0.68772 0.5 0.5 0.5
i18 i35 i28 i25 i30 i27 i31
similarity: 0.80146 0.73308 0.67651 0.50133 0.5 0.5
i19 i08 i03 i38 i06 i30 i31
similarity: 0.78709 0.75592 0.75567 0.50475 0.50414 0.5011


i20 i23 i40 i34 i37 i27 i31
similarity: 0.71314 0.6478 0.62419 0.50691 0.5 0.5
i21 i02 i17 i34 i30 i37 i31
similarity: 0.78411 0.73168 0.69789 0.50169 0.50128 0.50125
i22 i34 i39 i36 i27 i06 i31
similarity: 0.6676 0.65921 0.65254 0.50566 0.50369 0.5
i23 i40 i20 i28 i30 i31 i37
similarity: 0.71845 0.71314 0.6986 0.5 0.5 0.5
i24 i36 i03 i08 i06 i27 i31
similarity: 0.78231 0.77988 0.77725 0.50259 0.50084 0.5
i25 i17 i28 i21 i31 i30 i37
similarity: 0.76396 0.69132 0.68951 0.50276 0.50166 0.50157
i26 i33 i32 i28 i31 i15 i27
similarity: 0.57547 0.5724 0.54141 0.50261 0.50184 0.5


i27 i30 i32 i01 i23 i26 i35
similarity: 0.66179 0.5347 0.52746 0.5 0.5 0.5
i28 i18 i23 i25 i30 i31 i27
similarity: 0.73308 0.6986 0.69132 0.50824 0.50129 0.50053
i29 i30 i10 i01 i15 i17 i23
similarity: 0.73794 0.67378 0.63131 0.50071 0.5 0.5
i30 i29 i27 i16 i15 i17 i23
similarity: 0.73794 0.66179 0.59903 0.5003 0.5 0.5
i31 i06 i39 i11 i22 i23 i24
similarity: 0.51218 0.50606 0.5059 0.5 0.5 0.5
i32 i04 i03 i24 i06 i31 i15
similarity: 0.61618 0.61483 0.61338 0.50364 0.5005 0.50021
i33 i40 i28 i25 i04 i27 i31
similarity: 0.68941 0.64862 0.60476 0.50994 0.50803 0.50442


i34 i02 i39 i21 i32 i27 i31
similarity: 0.72618 0.71133 0.69789 0.51682 0.50444 0.50208
i35 i18 i28 i25 i04 i31 i27
similarity: 0.80146 0.66278 0.63476 0.50415 0.50077 0.5
i36 i38 i03 i24 i07 i15 i31
similarity: 0.83614 0.8155 0.78231 0.50593 0.50414 0.50288
i37 i36 i24 i32 i15 i17 i23
similarity: 0.60353 0.59869 0.59729 0.50036 0.5 0.5
i38 i36 i03 i08 i27 i06 i31
similarity: 0.83614 0.81654 0.76859 0.50389 0.50347 0.50026
i39 i02 i14 i34 i37 i26 i31
similarity: 0.73663 0.72073 0.71133 0.50814 0.50791 0.50606
i40 i07 i23 i09 i27 i04 i31
similarity: 0.76591 0.71845 0.69787 0.51035 0.50982 0.50363

Gross Texture Matching

Overview

The algorithm to compare images strictly on their total 'edginess' distribution, or in other words texture, is very similar to my color matching algorithm above, with a few notable differences.

1. For each image in the directory:
    a. Load the image with OpenCV2 (and append to a list)
    b. **Convert to grayscale image**
    c. **Convert to Laplacian image**
    d. **Generate a 1D histogram based on the Laplacian image, which is calculated for each pixel by multiplying that pixel's value by the number of neighbors and subtracting the sum of its neighbors pixels** (and append that to a list)
    e. *Optional*: Visualize the histogram as a bar graph (and append to a list)
        i. This actually looked really bad
2. Calculate the distance between every image pair (and store in a lookup dictionary)
    a. Use the histograms values of each image pair to determine the L1 norm
3. For each image:
    a. Determine the three images *most like it* and the three images *most unlike it* in terms of **edginess** distribution
    b. Display the septuples: original, three alike, three unalike
4. Determine group of four *most alike* images and group of four *most unalike* images
    a. Display the two groups of four: most alike, most unalike

All that gray stuff I already did. The differences in these algorithms however are notable, because they are very different ways to gauge the similarity between two images. Before, we looked at color distribution. Now, we look away from the color and at measures of "edginess." Because of this difference, I had to reassess my existing constants for bin size and black threshold, and consider new bin sizes and new thresholds for the Laplacian image.

Bin Size

Below are the bin size I decided on for my Laplacian image.

```
BLK_THRESH = 40
LAP_BINS = 128
LAP_BIN_SZ = int(COL_RANGE*8)/LAP_BINS
```

In the process of finding the edge distribution in an image, the first step was to make the image grayscale – that is, divide each RGB value by 3 and reduce it to a 1D histogram later. The was a simple matter, as you can see from the code below.

```
def grayscale(image, title):
    """Convert image into black and white with (R+G+B)/3"""
    gray = np.zeros(shape=(IMG_H, IMG_W))
    for row in xrange(IMG_H):
        for col in xrange(IMG_W):
            # compose gray image pixel by pixel
            pixel = image[row][col]
            rgb = int(pixel[0]) + int(pixel[1]) + int(pixel[2])
            g = int(round((rgb / 3.0),0)) # round up
            gray[row][col] = g
    # gray_img = cv2.imread(gray, cv2.IMREAD_UNCHANGED)
    cv2.imwrite('./gray/'+title+'.png', gray)
    return gray
```

Since I no longer had to worry about a cubic order of growth for Laplacian bin size as we were dealing with a 1D and no longer a 3D histogram, but still felt no need to sample all of the 256 range, I settled for a Laplacian bin size of 128. Through trial and error, I had realized that my original bin size of 8 was far too small, so I was happy with the final results after much tweaking with the Laplacian bin size and dealing with the black background in various ways.

Black Threshold

blk thresh: 0 15 20 40

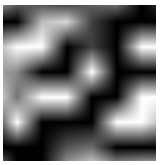
Dealing with the black background in this instance was involved trial and error as well. As you can see in the transformations above, these are the Laplacian transformations depending on black threshold. A threshold of 0 maintains the black background, whereas a threshold of 15 was too little and still kept black edges in the corner. I ended up staying with my original black threshold of 40, which makes sense as before I was making sure every RGB value was above 40 before including it in the histogram – here, since I had already divided RGB by 3 for to make the image grayscale, so comparing it to the cutoff of 40 once makes sense. The other choice I made was not to remove the black pixels when I made the grayscale or laplacian image. That was because I planned to save the laplacian image, and I didn't like the white background, and furthermore thought that might bring additional complications – which I tried actually, and it did you – you can see the commented out code down below. Instead, I treated the black threshold in the same was as my color histogram – when generating my 1D texture histogram, I only added the pixel count to the histogram if that gray pixel was above the black threshold.

```
def texture_histogram(lap, gray, title):
    hist = np.zeros(shape=LAP_BINS)
    bins = []
    for i in range(IMG_H):
        for j in range(IMG_W):
            pixel = lap[i][j]
            # Only count pixels outside the black threshold
            # to avoid image similarity due to shared black background
            if gray[i][j] > BLK_THRESH:
                if pixel < 256:
                    bin = abs(pixel/LAP_BIN_SZ)
                    hist[bin] += 1
                    if bin not in bins:
                        bins.append(bin)
    # plot = visualize_thist(gray, hist, bins, title)
    # print 'hist', hist
    # return hist, plot
    return hist
```

Laplacian Image

The sum of the Laplacian image following the grayscale conversion is an interesting measuring point. To generate the laplacian image, I create a new numpy array of 0s, and replace each original gray pixel with the formula:

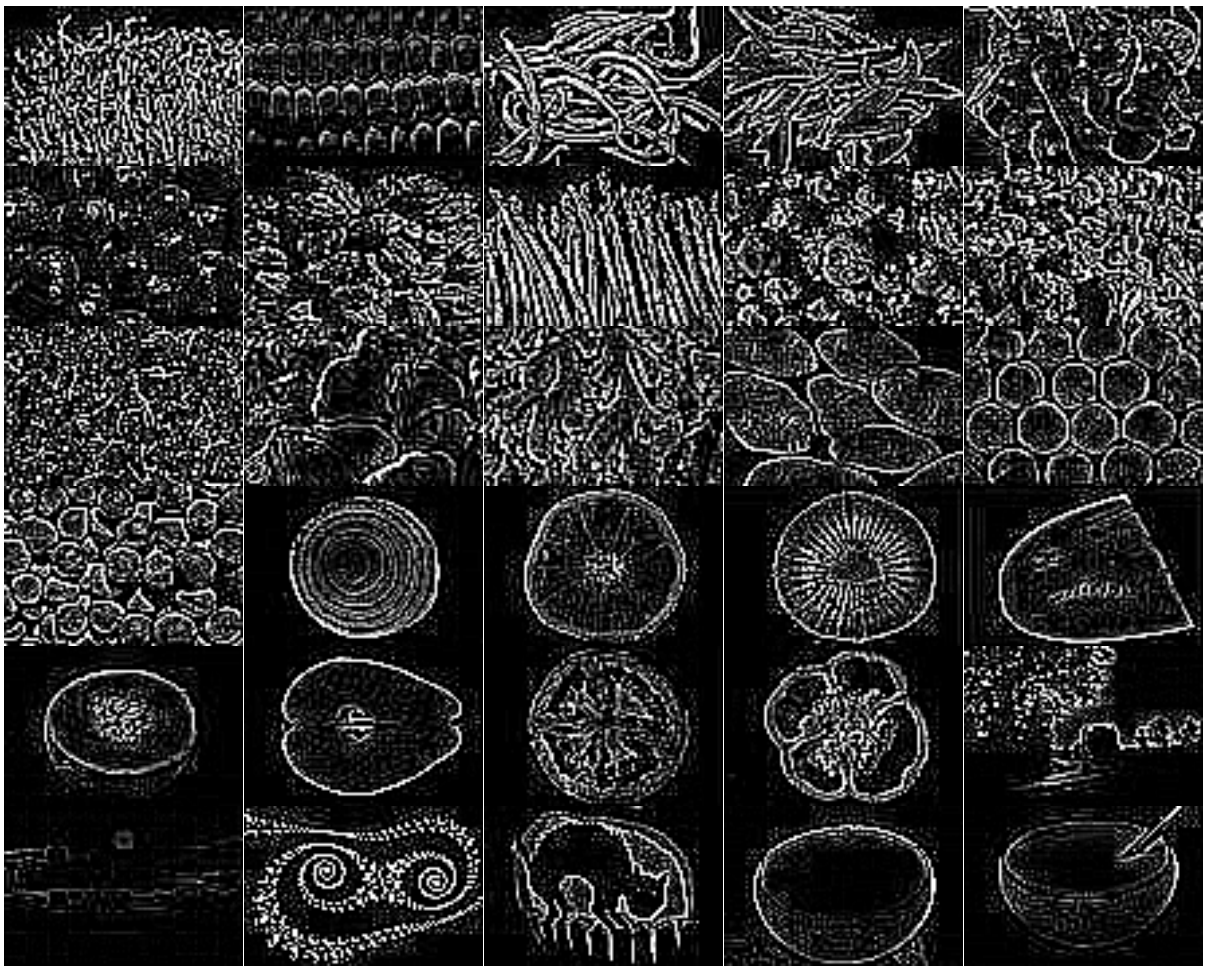
lapl_pixel = gray_pixel * num_of(neighbors) / ∑ (neighboring pixels)

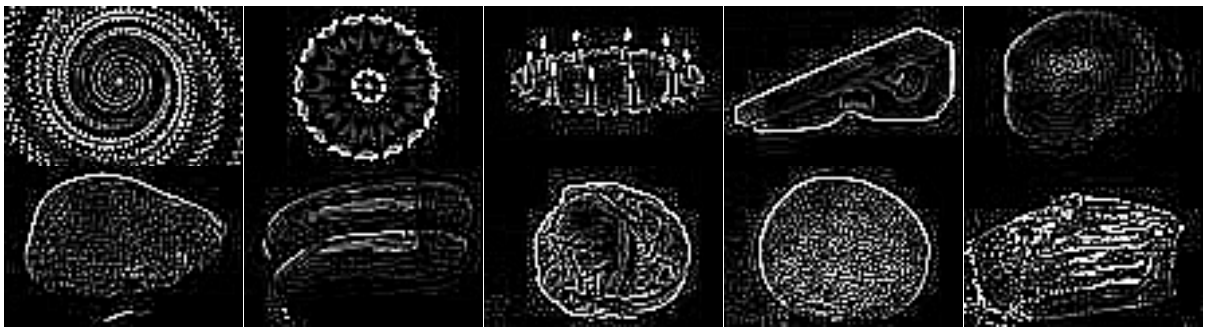
(i-1,j-1)	(i-1,j)	(i-1,j+1)		Top left	Top center	Top right
(i,j-1)	(i,j)	(i,j+1)	**i21 pixels**	Center right	Center	Center left
(i+1,j-1)	(i+1,j)	(i+1,j+1)		Bottom right	Bottom center	Bottom left

```
def laplacian(gray, title):
    """Make laplacian image by calculating sum of neighbors for every pixel"""
    # gray = grayscale(image)
    laplacian = np.zeros(shape=(IMG_H, IMG_W))
    # laplacian.fill(COL_RANGE) # white would be 255+255+255/3 = 255
```

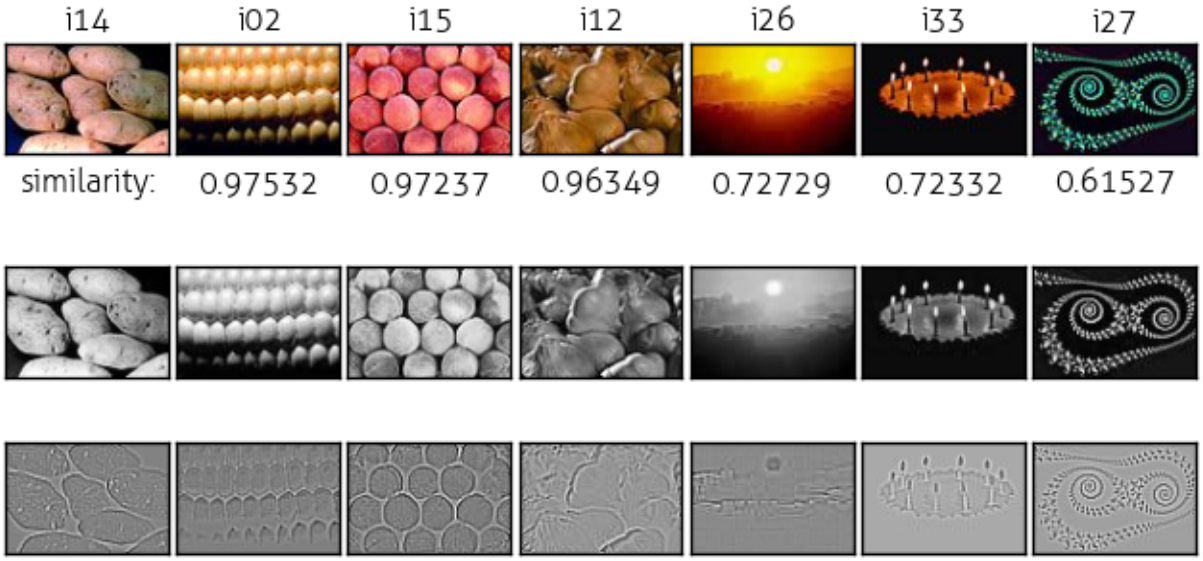
```
    for i in xrange(0,IMG_H):
        for j in xrange(0,IMG_W):
            # if gray[i][j] > int(BLK_THRESH/2):
            if gray[i][j]:
                neighbors = []
                for row in xrange(i-1,i+2):
                    for col in xrange(j-1,j+2):
                        if not (row is i and col is j) and index_valid(row,col):
                            neighbors.append(gray[row][col])
                neighbor_sum = reduce(lambda x, y: x+y, neighbors)
                n = len(neighbors)
                laplacian[i][j] = gray[i][j]*n - neighbor_sum
    # print 'laplacian',laplacian
    cv2.imwrite('./laplacian/'+title+'.png', laplacian)
    return laplacian

def index_valid(row,col):
    if (row < 0 or col < 0 or row >= IMG_H or col >= IMG_W):
        return False
    return True
```

The is near 0 values where there are homogenous colors (black in the pictures below, gray in the matplotlib display values as you can see in Appendix C), and traced edges.





One thing I noticed when trying to match texture histograms and analyzing the results is that the similarity between objects is often quite high. For instance, all these closely matching values are above 96%. The least similarity generally does not drop below 60%. The edge detection seems to work quite well however. Here we can see potatos grouped with corn and some other small round root plant. As these items are all round and bulbous, regardless of their proximity to size, their texture certain seems yo match quite well.



I also tried to account for more adjustments like changing the black threshold, and fiddling with the Laplacian bin size, but the images have tended to stay in that range, with 0.6% similarity considered very far. This is a good thing, although I wish the metric would be more sensitive to difference and distance between images.

As you can see in this example, which is even clearer in the bottomost Laplacian image, the clustering of the corn kernels matches the porous potatos and the little peaches(?) in i15 and mushrooms(?) in i12. I am surprised that the mushrooms aren't a closer match to the potatoes than the corn, seeing as they are similar in size from the vantage of the thumbnail, while the corn texture is more tightly clustered. The sunset is almost completely devoid of edges, which makes sense that it is considered unlike the other images.

Results

Four Best Match:

Four Worst Match:

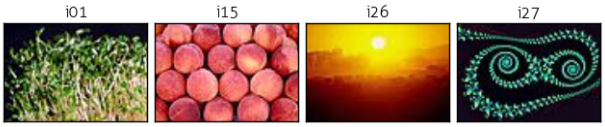
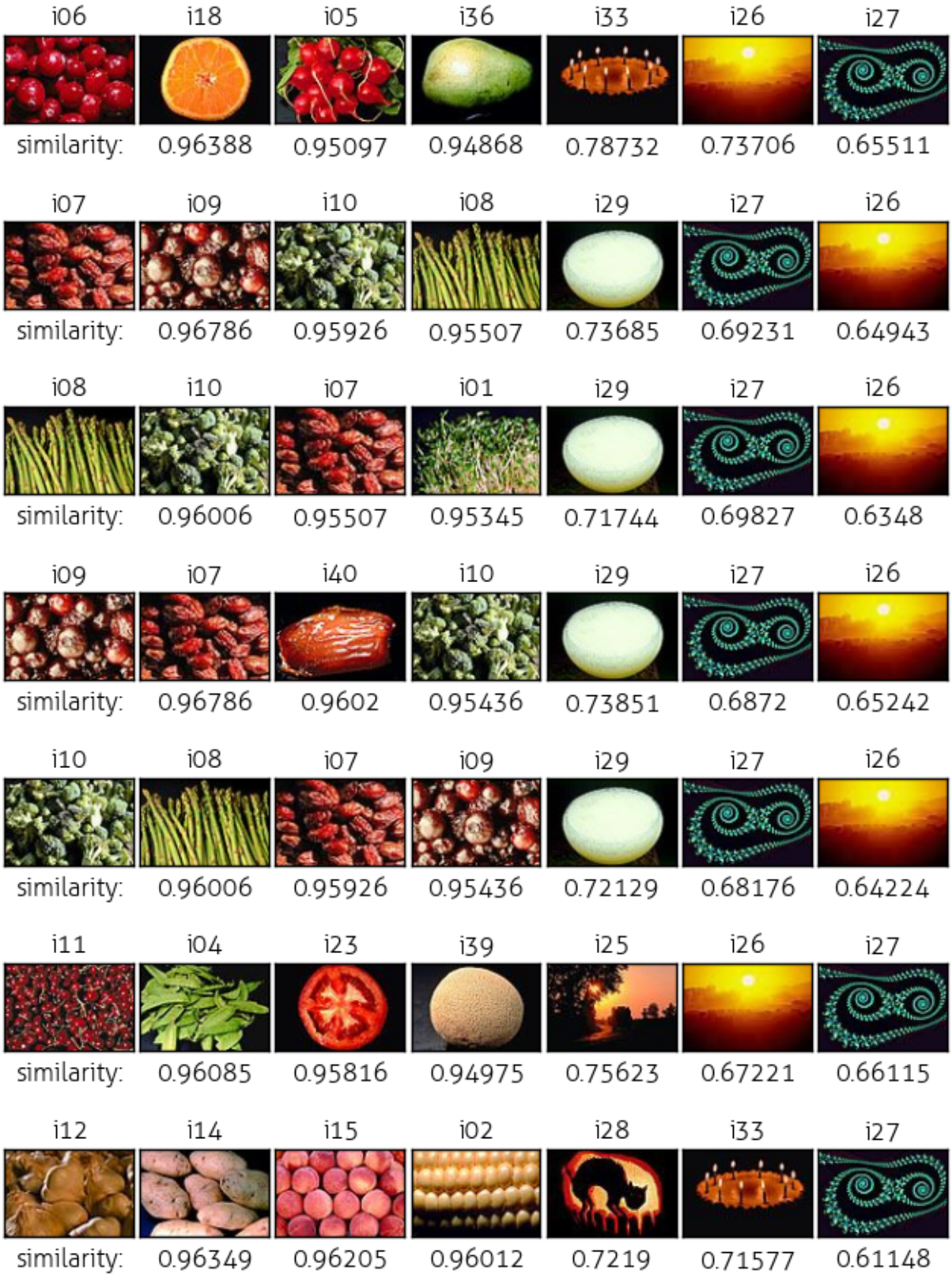
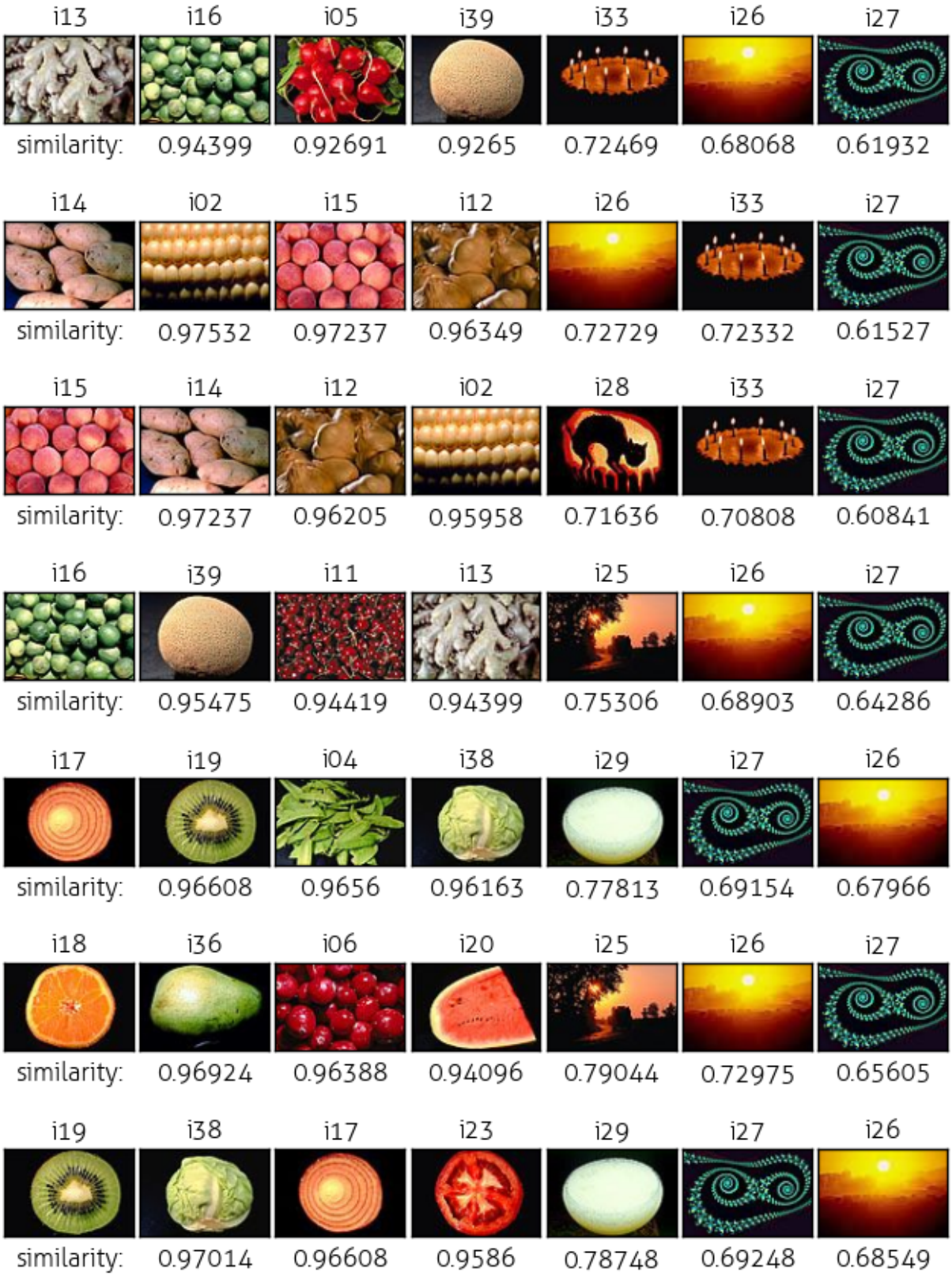
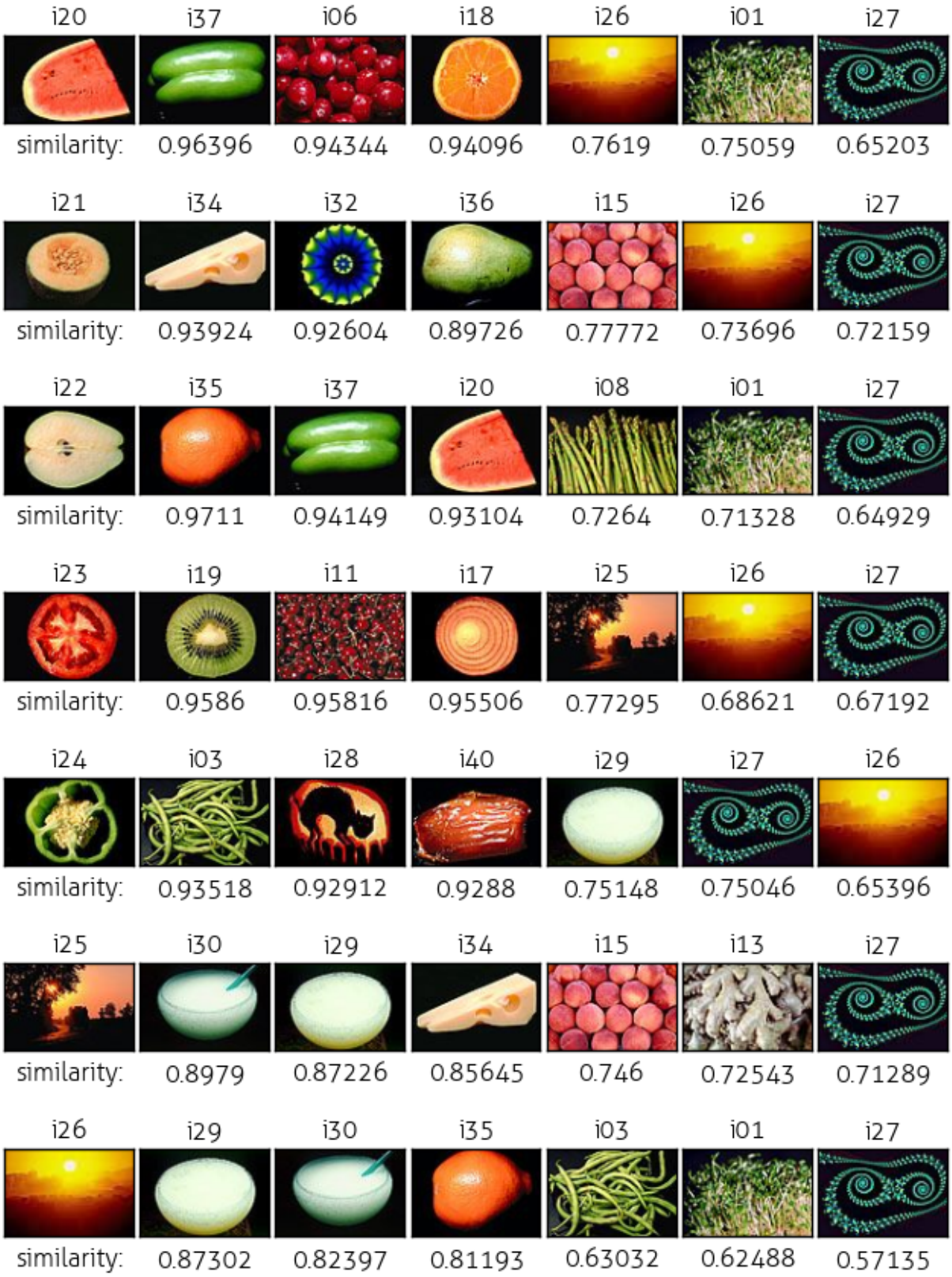


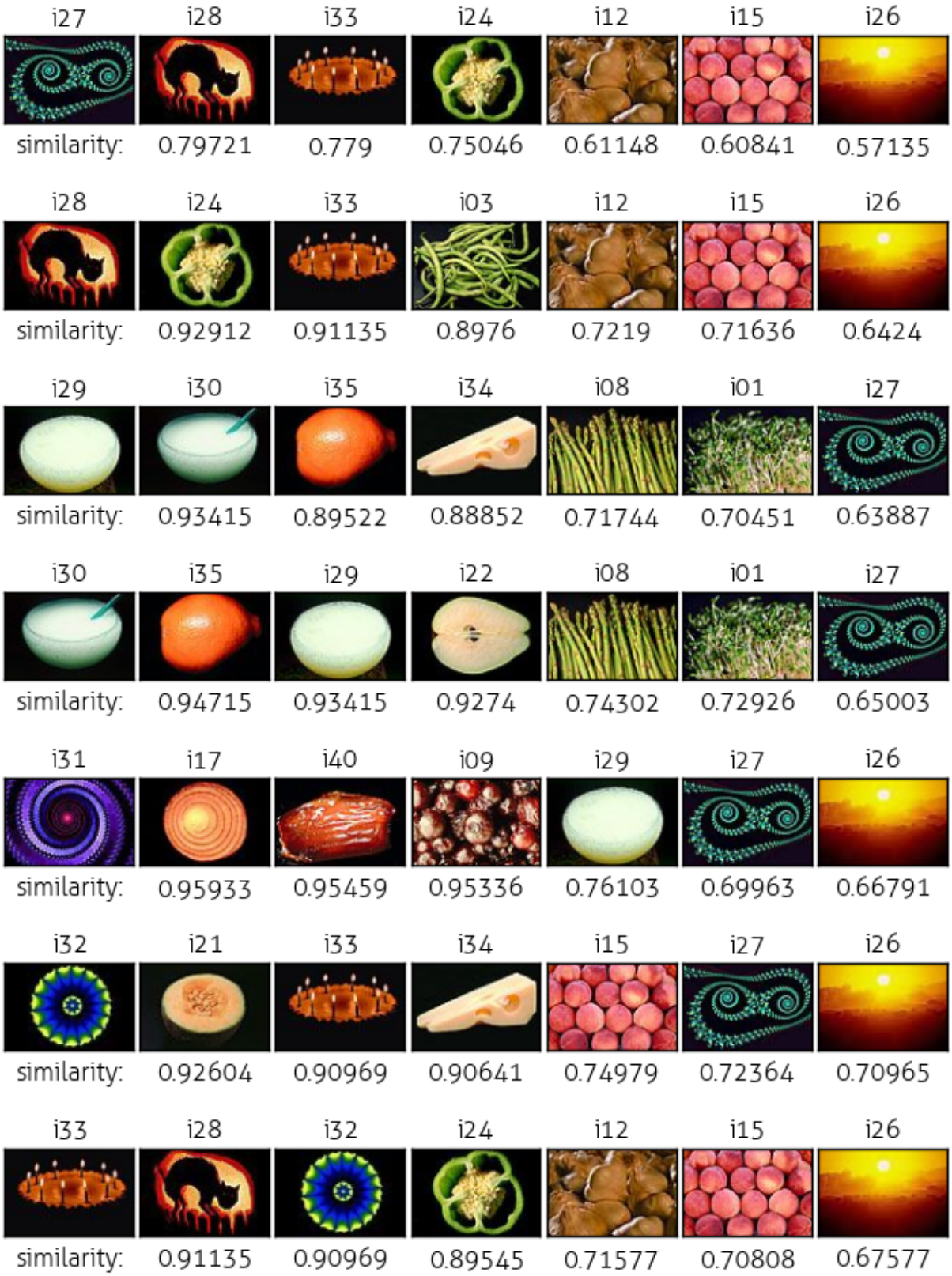
Image Septuples:

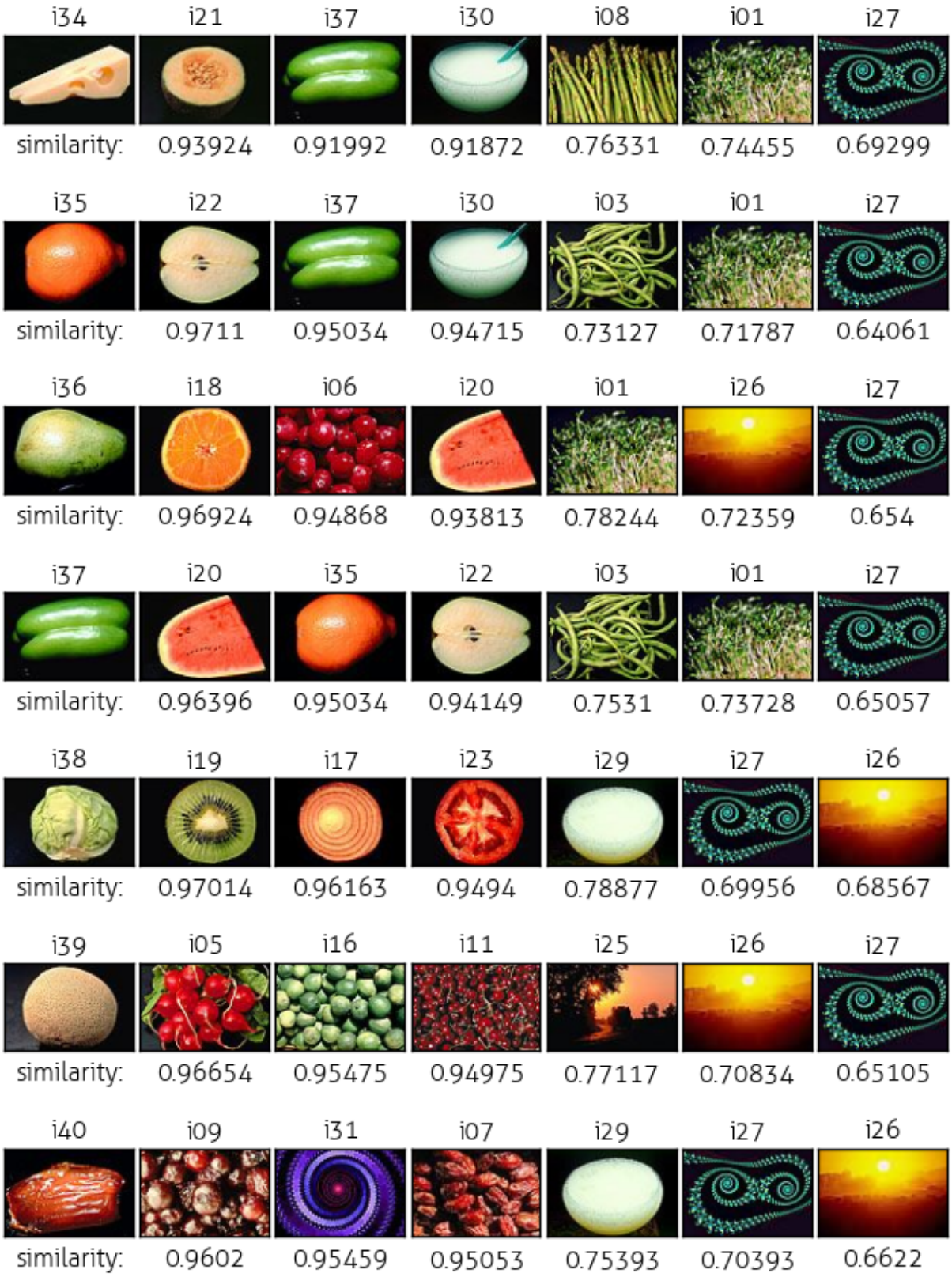
Original	Most Like	2nd Most Like	3rd Most Like	3rd Most Unlike	2nd Most Unlike	Most Unlike
i01	i08	i10	i07	i29	i27	i26
similarity:	0.95345	0.95024	0.94418	0.70451	0.6892	0.62488
i02	i14	i12	i15	i26	i33	i27
similarity:	0.97532	0.96012	0.95958	0.73995	0.73529	0.62141
i03	i08	i24	i07	i22	i29	i26
similarity:	0.94581	0.93518	0.93194	0.72722	0.71769	0.63032
i04	i17	i11	i31	i29	i27	i26
similarity:	0.9656	0.96085	0.95158	0.7642	0.67651	0.67184
i05	i39	i06	i16	i33	i26	i27
similarity:	0.96654	0.95097	0.94161	0.76549	0.71597	0.64429


i06 i18 i05 i36 i33 i26 i27
similarity: 0.96388 0.95097 0.94868 0.78732 0.73706 0.65511
i07 i09 i10 i08 i29 i27 i26
similarity: 0.96786 0.95926 0.95507 0.73685 0.69231 0.64943
i08 i10 i07 i01 i29 i27 i26
similarity: 0.96006 0.95507 0.95345 0.71744 0.69827 0.6348
i09 i07 i40 i10 i29 i27 i26
similarity: 0.96786 0.9602 0.95436 0.73851 0.6872 0.65242
i10 i08 i07 i09 i29 i27 i26
similarity: 0.96006 0.95926 0.95436 0.72129 0.68176 0.64224
i11 i04 i23 i39 i25 i26 i27
similarity: 0.96085 0.95816 0.94975 0.75623 0.67221 0.66115
i12 i14 i15 i02 i28 i33 i27
similarity: 0.96349 0.96205 0.96012 0.7219 0.71577 0.61148


i13 i16 i05 i39 i33 i26 i27
similarity: 0.94399 0.92691 0.9265 0.72469 0.68068 0.61932
i14 i02 i15 i12 i26 i33 i27
similarity: 0.97532 0.97237 0.96349 0.72729 0.72332 0.61527
i15 i14 i12 i02 i28 i33 i27
similarity: 0.97237 0.96205 0.95958 0.71636 0.70808 0.60841
i16 i39 i11 i13 i25 i26 i27
similarity: 0.95475 0.94419 0.94399 0.75306 0.68903 0.64286
i17 i19 i04 i38 i29 i27 i26
similarity: 0.96608 0.9656 0.96163 0.77813 0.69154 0.67966
i18 i36 i06 i20 i25 i26 i27
similarity: 0.96924 0.96388 0.94096 0.79044 0.72975 0.65605
i19 i38 i17 i23 i29 i27 i26
similarity: 0.97014 0.96608 0.9586 0.78748 0.69248 0.68549


i20 i37 i06 i18 i26 i01 i27
similarity: 0.96396 0.94344 0.94096 0.7619 0.75059 0.65203
i21 i34 i32 i36 i15 i26 i27
similarity: 0.93924 0.92604 0.89726 0.77772 0.73696 0.72159
i22 i35 i37 i20 i08 i01 i27
similarity: 0.9711 0.94149 0.93104 0.7264 0.71328 0.64929
i23 i19 i11 i17 i25 i26 i27
similarity: 0.9586 0.95816 0.95506 0.77295 0.68621 0.67192
i24 i03 i28 i40 i29 i27 i26
similarity: 0.93518 0.92912 0.9288 0.75148 0.75046 0.65396
i25 i30 i29 i34 i15 i13 i27
similarity: 0.8979 0.87226 0.85645 0.746 0.72543 0.71289
i26 i29 i30 i35 i03 i01 i27
similarity: 0.87302 0.82397 0.81193 0.63032 0.62488 0.57135


i27 i28 i33 i24 i12 i15 i26
similarity: 0.79721 0.779 0.75046 0.61148 0.60841 0.57135
i28 i24 i33 i03 i12 i15 i26
similarity: 0.92912 0.91135 0.8976 0.7219 0.71636 0.6424
i29 i30 i35 i34 i08 i01 i27
similarity: 0.93415 0.89522 0.88852 0.71744 0.70451 0.63887
i30 i35 i29 i22 i08 i01 i27
similarity: 0.94715 0.93415 0.9274 0.74302 0.72926 0.65003
i31 i17 i40 i09 i29 i27 i26
similarity: 0.95933 0.95459 0.95336 0.76103 0.69963 0.66791
i32 i21 i33 i34 i15 i27 i26
similarity: 0.92604 0.90969 0.90641 0.74979 0.72364 0.70965
i33 i28 i32 i24 i12 i15 i26
similarity: 0.91135 0.90969 0.89545 0.71577 0.70808 0.67577


i34 i21 i37 i30 i08 i01 i27
similarity: 0.93924 0.91992 0.91872 0.76331 0.74455 0.69299
i35 i22 i37 i30 i03 i01 i27
similarity: 0.9711 0.95034 0.94715 0.73127 0.71787 0.64061
i36 i18 i06 i20 i01 i26 i27
similarity: 0.96924 0.94868 0.93813 0.78244 0.72359 0.654
i37 i20 i35 i22 i03 i01 i27
similarity: 0.96396 0.95034 0.94149 0.7531 0.73728 0.65057
i38 i19 i17 i23 i29 i27 i26
similarity: 0.97014 0.96163 0.9494 0.78877 0.69956 0.68567
i39 i05 i16 i11 i25 i26 i27
similarity: 0.96654 0.95475 0.94975 0.77117 0.70834 0.65105
i40 i09 i31 i07 i29 i27 i26
similarity: 0.9602 0.95459 0.95053 0.75393 0.70393 0.6622

Combine Similarities, and Cluster

For this part, the linear sum for combined similarity is as follows:

$$combined\_similarity = r * (t\_sim) + (1 - r)*(c\_sim)$$

where I tested several values of the ratio r. I tested primarily by checking out how different values of r would change the most alike and most unalike group of four, and also comparing the clustering results for single link and complete link clustering.

Similarity Measure

The code below is a bit roundabout, because it is a similarity measure, but at the same time I have been using distance measures throughout so I ended up converted the distances to similarities, then converting that combined similarity back to distances. Using distances was also much more intuitive and convenient for the clustering method.

```
def combine_similarities(chist_dis, thist_dis):
    similarities = {}
    distances = {}
    closest = 1
    r = 0.5
    for i in xrange(NUM_IM):
        for j in xrange(i+1, NUM_IM):
            if (i,j) not in similarities:
                t_sim = 1 - thist_dis[(i,j)]
                c_sim = 1 - chist_dis[(i,j)]
                s = r*(t_sim) + (1-r)*(c_sim)
                d = 1 - s
                # d = r*thist_dis[(i,j)] + (1-r)*chist_dis[(i,j)]
                similarities[(i,j)] = s
                distances[(i,j)] = d
                # if (d < closest):
                #     closest = (i,j)
    return similarities, distances#, closest
```

Combined Similarity Results

Most Alike **Most Unalike**

r = 0, all color

As the lynchpins, I included the group of four based wholly on texture and wholly on color on either ends of the comparison of r values here. It appears that image i07 and i26 are very different texture wise (the former being very uniquely textures and the latter having nerly no texture at all), so they are mainstays. Oddly enough, and surprised I didn't notice this before, the round plums are both in the group of four most alike and most unalike.

At any rate, since it is harder to gauge how "unlike" two images are, for me at least, I looked mainly at the "alike" set to consider which r is better. While I originally first tested a value of r = 0.2 because I considered color to be far more important to my conception of visual similarity than texture, it turns out that the flip end r = 0.8 has better alike results. While it is easy to match items based on color similarity, texture turns out to be a good tiebreaker for differentiating those similar-color things.

Complete Link vs. Single Link Clustering

To create the 7 clusters, I followed the assignment directives of using "complete link" and then "single link" to cluster the images. For both, all images begin in their own cluster or disjoint sets, and each pair is compared to find the nearest pair according to the new combined similarity. Since I had generated pairwise distances in that method already, I passed in the lookup table and just iterated through every cluster pair to find the nearest pair. The difference between complete link and single link comes when dealing with grouping multi-element clusters. In a complete link model, the "nearness" of wo clusters is the fartest distance between any 2 elements in those 2 clusters. In a single link model, the nearest distance between two elements in those clusters is what is considered the nearness

My implementation is on the next page.

```
def cluster(distances, link):
    clusters = {}
    for idx in xrange(0,NUM_IM):
        clusters[(idx,)] = (idx,)
    # print clusters
    counter = 0
    while len(clusters) > NUM_CLUSTERS:
        nearest_pair = (None, None)
        nearest_dist = 1 # total dissimilarity
        counter += 1
        # Go through every cluster-pair to find nearest pair
        for a in clusters:
            for b in clusters:
                # Do not compare with same cluster
                if a is not b:
                    dist = link
                    # For each element in both clusters, determine "nearness"
                    # Complete nearness: farthest distance bt any 2 el in 2 clusters
                    # Single nearness: nearest distance bt any 2 el in 2 clusters
                    for i in clusters[a]:
                        for j in clusters[b]:
                            k = (i,j)
                            if i > j:
                                k = (j,i) # tuples always in ascending order
                            curr_dist = distances[k]
                            if (link is 0 and curr_dist > dist) or \
                            (link is 1 and curr_dist < dist):
                                dist = curr_dist
                    # Find out if this is the nearest pair so far in the iteration
                    if dist < nearest_dist:
                        nearest_dist = dist
                        nearest_pair = (a,b)
                        nearest_pair_values = clusters[a]+clusters[b]
        # combine the nearest pair of clusters and remove old clusters
        clusters.pop(nearest_pair[0])
        clusters.pop(nearest_pair[1])
        # new_pair = nearest_pair[0] + nearest_pair[1]
        clusters[nearest_pair_values] = nearest_pair_values
        # clusters[new_pair] = clusters[new_pair]
        print counter, clusters.keys()

    # print clusters
    clusters = clusters.keys()
    return clusters
```

I had several false starts with storing the clustered values, but I finally was able to verify that this clustering code wotks by testing the small 4 element example provided in the assignment specification. I have included the console output of the clustering process and hope you find it illustrative to my method.

```
1 : New distance for (2,) (3,) 0 -> 0.4
1 : Replace distance with  ((2,), (3,)) 1000 -> 0.4
1 : New distance for (2,) (1,) 0 -> 0.5
```

```
1 : New distance for (2,) (4,) 0 -> 0.6
1 : New distance for (3,) (2,) 0 -> 0.4
1 : New distance for (3,) (1,) 0 -> 0.1
1 : Replace distance with  ((3,), (1,)) 0.4 -> 0.1
1 : New distance for (3,) (4,) 0 -> 0.3
1 : New distance for (1,) (2,) 0 -> 0.5
1 : New distance for (1,) (3,) 0 -> 0.1
1 : New distance for (1,) (4,) 0 -> 0.2
1 : New distance for (4,) (2,) 0 -> 0.6
1 : New distance for (4,) (3,) 0 -> 0.3
1 : New distance for (4,) (1,) 0 -> 0.2
1 [(2,), (3, 1), (4,)]
2 : New distance for (2,) (3, 1) 0 -> 0.4
2 : New distance for (2,) (3, 1) 0.4 -> 0.5
2 : Replace distance with  ((2,), (3, 1)) 1000 -> 0.5
2 : New distance for (2,) (4,) 0 -> 0.6
2 : New distance for (3, 1) (2,) 0 -> 0.4
2 : New distance for (3, 1) (2,) 0.4 -> 0.5
2 : New distance for (3, 1) (4,) 0 -> 0.3
2 : Replace distance with  ((3, 1), (4,)) 0.5 -> 0.3
2 : New distance for (4,) (2,) 0 -> 0.6
2 : New distance for (4,) (3, 1) 0 -> 0.3
2 [(2,), (3, 1, 4)]
3 : New distance for (2,) (3, 1, 4) 0 -> 0.4
3 : New distance for (2,) (3, 1, 4) 0.4 -> 0.5
3 : New distance for (2,) (3, 1, 4) 0.5 -> 0.6
3 : Replace distance with  ((2,), (3, 1, 4)) 1000 -> 0.6
3 : New distance for (3, 1, 4) (2,) 0 -> 0.4
3 : New distance for (3, 1, 4) (2,) 0.4 -> 0.5
3 : New distance for (3, 1, 4) (2,) 0.5 -> 0.6
3 [(2, 3, 1, 4)]
[(2, 3, 1, 4)]
```

0.0	0.5	0.1	0.2
0.5	0.0	0.4	0.6
0.1	0.4	0.0	0.3
0.2	0.6	0.3	0.0

The clustering process goes:

```
{{1}, {2}, {3}, {4}}
{{1,3}, {2}, {4}}
{{1,3,4}, {2}}
{{1,2,3,4}}
```

Results with different r-values

As can be expected, the complete link at each step creates clusters that are cliques (every image is more related to every other image) whereas single link at each step, as the name suggests, link at one spot and grow like a minimum spanning tree. While before I said the value of r = 0.8 was more successful, in this case the cluster created by the combined similarity with r = 0.2 looks better, and r = 0.5 looks best. I say this because I cannot really discern the relationships between the images in

the largest cluster in r = 0.8, possibly because they are dominated by texture and the images are so small in this particular display that I am relying mostly on color to group them myself.

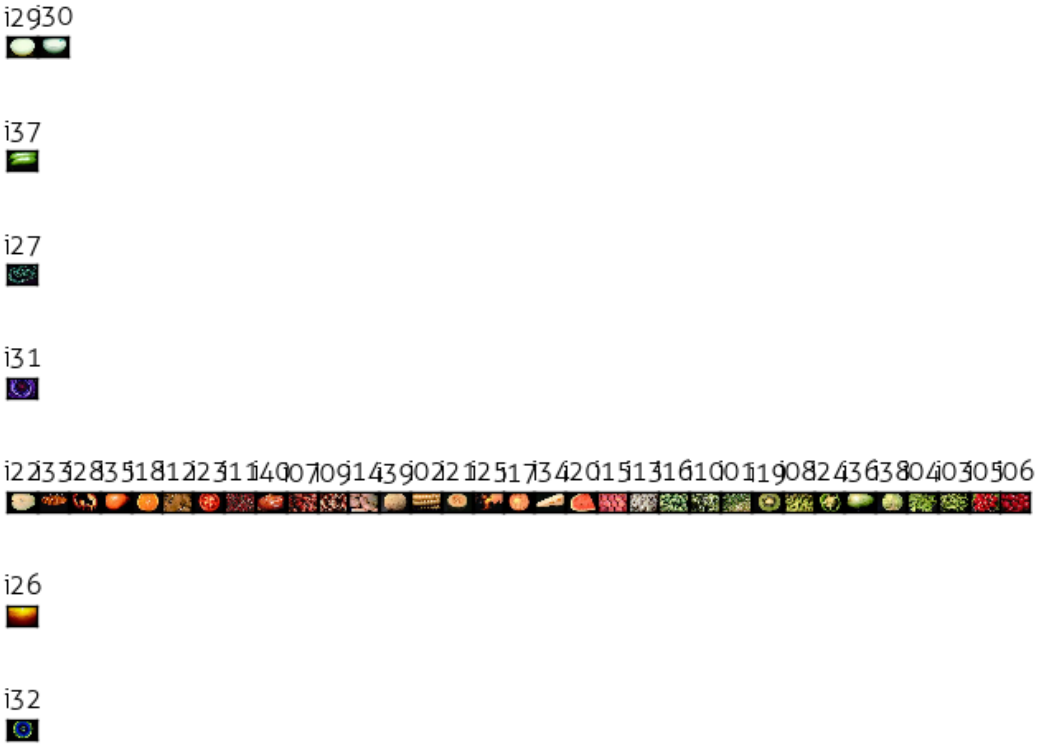
With r = 0.2, it is easy to see the clusters in the complete link: red, vivid green, creamy and light green, orange, and the outliers: the weird green fractal, purple vortex and generic sunset. If this was a pure color match, the sunset would porbbaly be included, but it's smoothness is enough to make it an outlier. As for the single link example, the images are indeed very small due to the long chain in the middle, but since the clusters were stored as arrays I believe they were appended to each other in a sequential fashion, which makes it convenient to view the single linkages between them: in this case a color spectrum from orange to yellow green to green and then two reds (which should be opposed to green, but appear to have the same globular shape).

r = 0.5 has similar results, notably the purple vortex has joined the shriveled plum due to their comparable texture, and no longer the outlier. I might even consider that cluster to be more successful than the r = 0.2 example. The complete link is in this case also better than the single link case, exhibiting the same clique (multi-connected, complete-linked) behavior. Another reason I might deem it more successful is that the groups are slightly more balanced in numbers. Something I noticed during my user study was that even though I didn't set down any rules for clustering sizes, people tended to try to group the images in similar sizes and not leave outliers alone. If I were to improve my clustering algorithm, I would perhaps do that, balance out the membership sizes of the clusters and also possibly place outliers together.

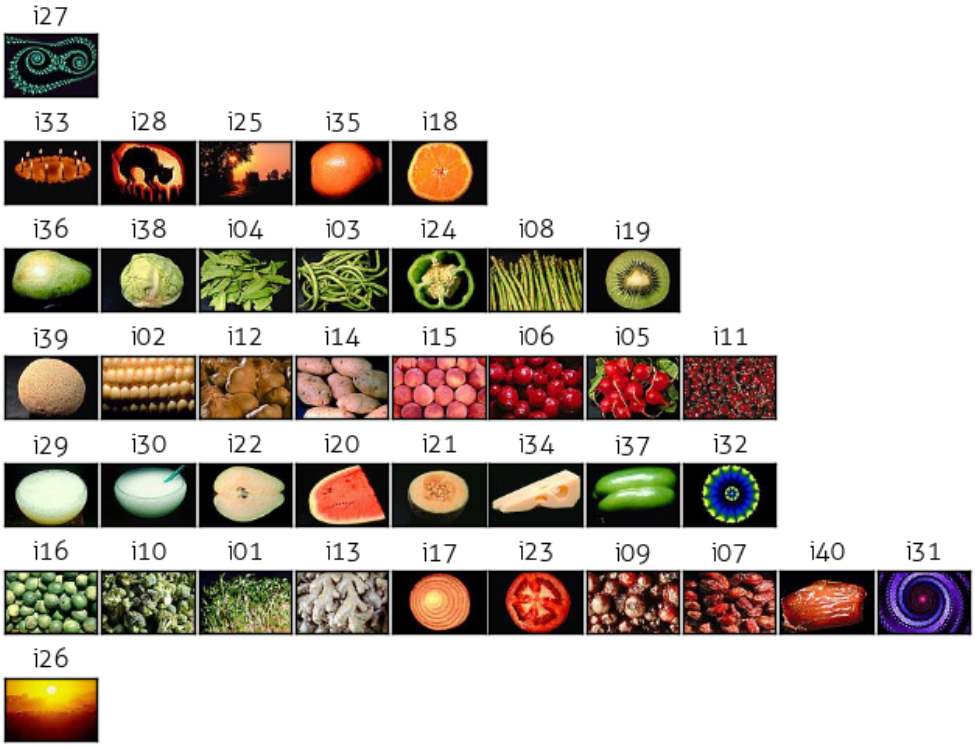
The clustering results are included below.

r = 0.2, when color dominates - complete link

r = 0.2, single link



r = 0.5, compete link



r = 0.5, single link

i29 i30

i37 i12 i25 i36 i38 i19 i24 i40 i08 i04 i07 i31 i10 i01 i16 i22 i20 i06 i05 i13 i39 i02 i14 i41 i21 i11 i09 i07 i40 i17 i23 i32 i34 i42 i03 i33 i28 i35 i18

i27

i31

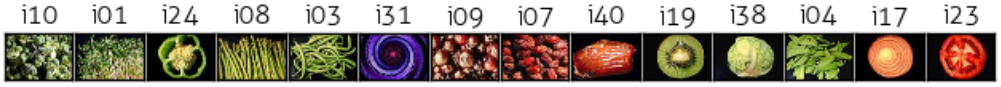
i15

i26

i32

r = 0.8, complete link

i10 i01 i24 i08 i03 i31 i09 i07 i40 i19 i38 i04 i17 i23



i27

i29 i30 i25

i33 i28 i21 i34 i32

i06 i36 i18 i22 i35 i37 i20

i16 i39 i05 i11 i13 i14 i12 i02 i15

i26

r = 0.8, single link

i33

i29 i30

i25

i27

i37 i20 i03 i11 i41 i12 i02 i21 i15 i16 i06 i39 i05 i17 i23 i09 i07 i14 i01 i13 i18 i19 i38 i04 i10 i00 i12 i40 i08 i03 i31 i32 i33 i35 i28 i21 i34

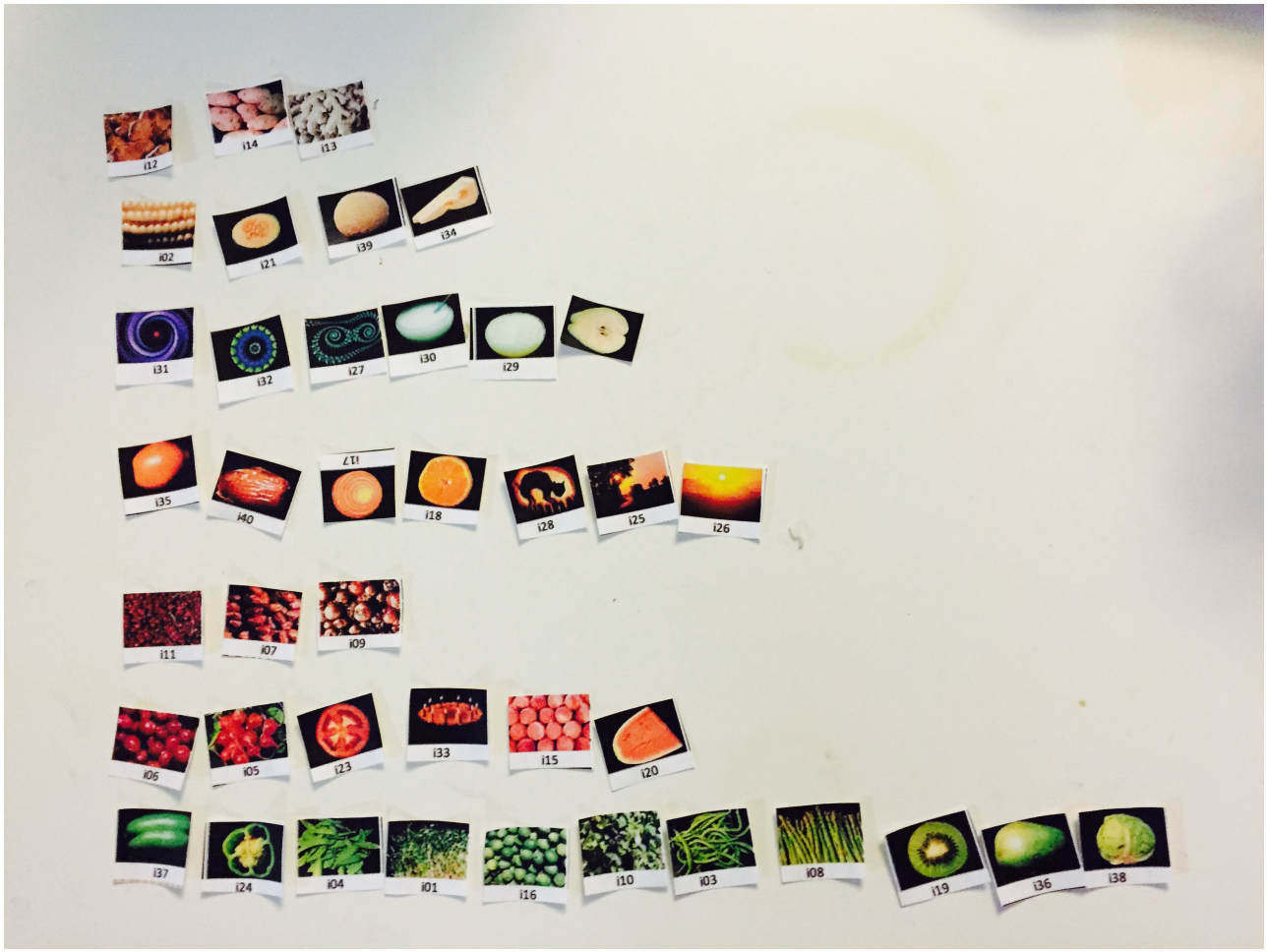
i26

i32

Creative Step

User Studies

For the final creative, evaluative step, I had some human users (ahem, four wonderful friends) match the best and worst image pairs by color and texture as an intelligent baseline against which I could measure my program's performance.



Funny note about human learning and knowledge sharing. The first of my willing users (Robert) took the longest time (over an hour), because he started from the beginning and answered the worst/best choices without organizing the pictures or clustering them. At the end, he commented that he wished he had done the clustering first. The next person, Jacky, finished much more quickly after doing clustering first in a more haphazard fashion (40 minutes). Alex, an artist, did clustering in a more organized fashion and finished in under 30 minutes. The photo above is his organized doing. Ashley, having seen everyone's struggles and heard complaints about how tedious the tasks were, and for some reason intrigued by the study even though I told her I already had enough responses, managed to finish in about 15 minutes. Their answer sheets are filed in Appendix A.

Performance Measure

To measure my system's results against my friend's decisions, I coded evaluation functions that took in the a human responses as a csv file, stored it as a list of values, and compared it to the system's results for image matching and image clustering. To measure accuracy and performance, I took advantage (perhaps too freely) the handy measures of precision, recall, their F1 harmonic mean and the Rand index for clustering. The formulas are as follows (informally defined, as you can expect from me by now):

$$precision = tp / tp + fp$$

$$recall = tp / tp + fn$$

$$f1 = (2.0 * precision * recall) / (precision + recall)$$

$$rand\_index = (tp + tn) / (tp + fp + fn + tn))$$

where

tp = true positive
fp = false positive
fn = false negtive
tn = true negative

measure different things depending on the evaluation. The two processes of evaluation and their respective results and discussed separately below.

Matching Evaluation

The matching evaluation considers these cases:

```
tp -- if human guess is in sys results
fn -- if human guess is missing from sys results
fp -- if human guess is in opposite of sys results
```

And counts their occurrences for 1) best color match, 2) worst color match, 3) best texture match, and 4) worst texture match. tp is important as a measure of how many times the system can get a human results, fn is whever the system is missing a human answer in its top 3 candidates, and fp is the notable case that issues a print statement every time the condition is met – when a human says something is the worst/best match for something, but the system has placed it in the opposite best/worst match category. I use the numbers crunched by this function to evaluate the system performance and how accurately the computer can simulate human choices in visual matching. As input, the function takes in the previous color, texture results, and compares to the the specific entered indvidual's answers for that category for every single image. Stay tuned, as I discuss how the system (and human!) performance after the break.

```
def match_eval(cresults, tresults, hresults):
    # best,worst color and best,worst texture measures
    bc_tp, bc_fp, bc_fn = 0,0,0
    wc_tp, wc_fp, wc_fn = 0,0,0
    bt_tp, bt_fp, bt_fn = 0,0,0
    wt_tp, wt_fp, wt_fn = 0,0,0

    counter = 0

    for i in xrange(0,NUM_IM*7,7):

        # range indices for best/worst system results
        b_from = i+1
        b_to = i+4
        w_from = i+4
        w_to = i+7

        # human results for best color, worst color,
        # best texture, worst texture
        # note: need var j because human list increments by 5
        # whereas system list increments by 7
        j = i - (2*counter)
        c_best = hresults[j+1]
        c_worst = hresults[j+2]
        t_best = hresults[j+3]
        t_worst = hresults[j+4]
        counter += 1

        # check best color match
        if c_best in cresults[b_from:b_to]:
            bc_tp += 1
        else:
            bc_fn += 1
        # since we are measuring computer against human perf:
        # program messed up and made a big false positive
        if c_best in cresults[w_from:w_to]:
            wc_fp += 1
            print 'Human best color match in system worst!', cresults[i]+1, c_best+1
            # +1 to make it readable - display current index and image in q

        # check worst color match
        if c_worst in cresults[w_from:w_to]:
            wc_tp += 1
        else:
            wc_fn += 1
        if c_worst in cresults[b_from:b_to]:
            bc_fp += 1
            print'Human worst color match in system best!', cresults[i]+1, c_worst+1

        # check best texture match
        if t_best in tresults[b_from:b_to]:
            bt_tp += 1
        else:
```

```
            bt_fn += 1
        # since we are measuring computer against human perf:
        # program messed up and made a big false positive
        if t_best in tresults[w_from:w_to]:
            wt_fp += 1
            print 'Human best texture match in system worst!', cresults[i]+1,t_best+1

        # check worst texture match
        if t_worst in tresults[w_from:w_to]:
            wt_tp += 1
        else:
            wt_fn += 1
        if t_worst in tresults[b_from:b_to]:
            bt_fp += 1
            print'Human worst texture match in system best!', cresults[i]+1,t_worst+1

    # Calculate total tp,fp,fn values
    tp = bc_tp + wc_tp + bt_tp + wt_tp
    fp = bc_fp + wc_fp + bt_fp + wt_fp
    fn = bc_fn + wc_fn + bt_fn + wt_fn

    print "1: BEST COLOR MATCH RESULTS"
    print_results(bc_tp, bc_fp, bc_fn)

    print "2: WORST COLOR MATCH RESULTS"
    print_results(wc_tp, wc_fp, wc_fn)

    print "3: BEST TEXTURE MATCH RESULTS"
    print_results(bt_tp, bt_fp, bt_fn)

    print "4: WORST TEXTURE MATCH RESULTS"
    print_results(wt_tp, wt_fp, wt_fn)

    print "5: OVERALL COLOR AND TEXTURE MATCH RESULTS"

def print_results(tp, fp, fn):
    print "TP: %d, FP: %d, FN: %d" % (tp, fp, fn)

    precision = float(tp) / (tp + fp)
    recall = float(tp) / (tp + fn)

    print "Precision : %f" % precision
    print "Recall    : %f" % recall
    print "F1        : %f" % ((2.0 * precision * recall) / (precision + recall))
```

Matching Performance

Overall, the evaluation was written with the intention of comparing the system performance to human baselines. It is interesting to also compare the other way and consider how closely humans can approximate these computational decisions with their eyes and the differing performance and agreements between them.

While the numbers in precision are nice, it was hard to consider what might really be a "wrong" or "false positive" answer, which I considered to mean if the human and computer judgments of best/worst were completely at ends with each other. If there were no such cases as that, then precision was just a measure of tp/tp+0, which is 1 and perfect. Recall for closest match is an interesting thing to look at it, and it appears that only half the time the computer can guess what people would put as the closest or further color match, with better results for closest.
Since the F1 values can be seen as a mean between precision and recall (quality and quantity), we can see that generally there is more agreement for colors than textures.

Indeed, texture posed quite the pose for most of my friends, who kept asking me how they could feel the texture of a 2D image and how you were supposed to compare similar and further texture. Even more than colors, textures appear to be a subjective and ambiguous area of comparison. So in terms of performance, judging from the numbers in the table below, this seemed to be the trajectory of better to less better performance:

Color Best > Color Worst > Texture Best > Texture Worst

	Robert		Jacky		Alex		Ashley	
Color	Best	Worst	Best	Worst	Best	Worst	Best	Worst
Precision	1.00000	0.94444	0.91667	1.00000	1.00000	0.83333	1.00000	1.00000
Recall	0.55000	0.42500	0.27500	0.15000	0.50000	0.25000	0.45000	0.22500
F1	0.70968	0.58621	0.42308	0.26087	0.66667	0.38462	0.62069	0.36735
Texture								
Precision	0.92308	0.89474	0.72727	1.00000	0.90909	0.92857	1.00000	0.70000
Recall	0.30000	0.42500	0.20000	0.20000	0.25000	0.32500	0.22500	0.17500
F1	0.45283	0.57627	0.31373	0.33333	0.39216	0.48148	0.36735	0.28000

Overall	Robert	Jacky	Alex	Ashley
Precision	**0.94444**	**0.89189**	**0.92982**	**0.93478**
Recall	0.42500	0.20625	0.33125	0.26875
F1	0.58621	0.33503	0.48848	0.41748

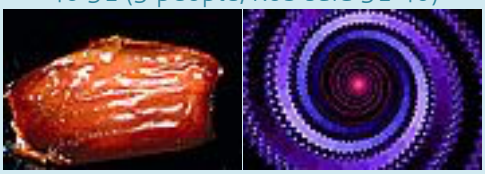
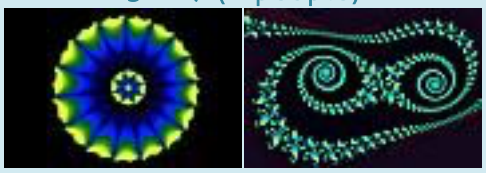
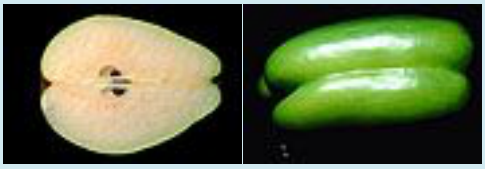
Interesting to see how people come up with different subjective answers and the system compares to them. Generally however, the values seem quite consistent between this subset of four individuals, with an average harmonic F1 mean of 50% (not very good, but could be better if we increased the top three group to top five perhaps for the system, and also tweaked the algorithm).

Below are the major alarm points that set off the false positive alarm – when human best matches the system's worst, or vice versa in some category. When I looked up the image indices, I had to agree that they were pretty confusing. Quickly, let's go through them.

Human best match in system worst:

The purple vortex seems to be confusing in terms of color and texture. For 31, 27 and 32, 27, it seems natural that people see it as similar in texture to the other fractal looking images, but the system considers them the worst match for texture. Similarly, it seems hard to find a close color for purple , while people saw complementary purple and yellow, the system considred them the worst match.

Also interesting is that while people might consider two images to be the worst pair match, if they see the same two images again, they might not necessarily think that way. For i17 i29 and, the concentric circles might not be so apparent but for the system they are completely different textures. The 23 and 27 pairing may be a case of matching color instead of exture.

Human best texture match in system worst!	Human worst texture match in system best!
31 27 (2 people) 	40 31 (3 people, Rob said 31 40) 
32 27 (2 people) 	22 37 
17 29 	38 19 
23 25 	

Human best color match in system worst!	Human worst color match in system best!
32 31 (2 people) 	31 6 
31 22 	

For human worst match in system best:
In the top case on the right, three people said the worst match for texture was a shriveled date and the purple vortex – but the computer says it is the best. It could have to do with our associations – what does a dried fruit have to do with this geometric shape? Below that, the system considers the best match a sliced apple and green bellpepper, but for me at leas tthis gives opposing notions of inside and outside. Same with the wrinkled cabbage and seedy kiwi interior. In terms of color, the purple vortex strikes again, while one of my friends thought purple and red were the worst color match, the system though they were one of the best matches (possibly due to the lack of blue and purple in the image series and overabundance of green).

Clustering Evaluation

Before I discuss ther results, let me briefly talk about my clustering performance measure. For the image clusters, I use the rand index, which I described a few pages up as

rand_index = (tp + tn) / (tp + fp + fn + tn))

In the case of this algorithm:
tp -- is defined as when a pair of images in the human result is in the same cluster, and the same pair of images is also in the same cluster in the system results
fp -- is when a pair of images share a cluster in the human groupings, but are in different clusters according to the system
fn -- is when the system clusters a pair of images, but the human has them in separate clusters
tn -- is when we consider a pair of images in different clusters in both the human and system results

The function takes in two lists of 7 tuples where each cluster is a tuple, one list generated by the system, the other by one of my friends. I wrote a helper function called cluster_id's that unpacks each of the 40 image elements to different list indices and stores in the element their cluster/set number for easy lookup to determine whether two images belong in the same set (just check syst[i] and syst[j] for instance!)

Using the rand index, I can more objectively re-evaluate the r-value for the combined similarity complete clusters and determine which one might have better performance after all.

```
def cluster_eval(system, human):
    # Change list of tuples -> list of lists
    hum = cluster_id(human, 1)
    syst = cluster_id(system, 0)

    # Initialize True-Pos, True-Neg False-Pos, False-Neg measures
    tp, tn, fp, fn = 0,0,0,0
    for i in xrange(NUM_IM):
        for j in xrange(i+1,NUM_IM):
            if hum[i] == hum[j] and syst[i] == syst[j]:
                tp += 1
            if hum[i] == hum[j] and syst[i] != syst[j]:
                fp += 1
            if hum[i] != hum[j] and syst[i] == syst[j]:
                fn += 1
            if hum[i] != hum[j] and syst[i] != syst[j]:
```

```
                tn += 1
    print "TP: %d, FP: %d, TN: %d, FN: %d" % (tp, fp, tn, fn)

    rand_idx = (float(tp + tn) / (tp + fp + fn + tn))
    precision = float(tp) / (tp + fp)
    recall = float(tp) / (tp + fn)

    # Print results:
    print "Rand index: %f" % rand_idx
    print "Precision : %f" % precision
    print "Recall    : %f" % recall
    print "F1        : %f" % ((2.0 * precision * recall) / (precision + recall))

    return rand_idx

def cluster_id(list_of_tups, key):
    '''
    Take list of cluster tups and return list of cluster id's
    where each list index is the image and the el is its set id
    key -- 1 if its starting index needs to be decremented to 0
    '''
    cluster_ids = [0] * 40
    for i in xrange(7):
        tup = list_of_tups[i]
        for j in tup:
            if key is 1:
                cluster_ids[j-1] = i
            else:
                cluster_ids[j] = i
    return cluster_ids
```

Clustering Performance

r = 0.2	Rob	Jacky	Alex	Ashley
Rand	**0.77692**	**0.78077**	**0.79103**	**0.77436**
Precision	0.45455	0.46875	0.50000	0.44444
Recall	0.33742	0.36810	0.36810	0.31902
F1	0.38732	0.41237	0.42403	0.37143
r = 0.5				
Rand	**0.79872**	**0.80000**	**0.78205**	**0.77051**
Precision	0.39669	0.40625	0.34167	0.29915
Recall	0.36364	0.39394	0.31061	0.26515
F1	0.37945	0.40000	0.32540	0.28112
r = 0.8				
Rand	**0.75128**	**0.75000**	**0.74231**	**0.71795**
Precision	0.36364	0.36719	0.33333	0.24786
Recall	0.27329	0.29193	0.24845	0.18012
F1	0.31206	0.32526	0.28470	0.20863

The Rand index gives good comparison for the different r values in the similarity comparison. It appears my seeing-based assessment of the cluster groupings and preference for the complete link clustered generated by r=0.5 can be backed up by the higher Rand index. Note that I did not use the single link clustering – while it was interesting to see that sort of trajectory of extending the cluster, it was quite unlike how humans typically organize clusters of objects so I did not bother evaluating it, and instead focused on seeing the differing performance of the r-values.

While the precision and recall methods seem quite subjective and I'm not completely sure this is the best metric of performance, the rand index is very interesting and informative as a measure of similarity between two data clusterings. It gives very consistent values for all the users, which means people were grouping the images in similar ways. While it is related to accuracy, it's more interesting to see how similar these clusters can be, and fun to think about how to improve this system so the Rand measure can be raised in terms of accuracy to render quasi intelligent human judgment on image similarities

Console Results for Evaluations

This is the more complete information of the exact number of TP, FP, FN's etc. generated each iteration of evaluation. I simply copied the most relevant and telling information above. It is interesting to note the differences btween human performance as well. On one hand, I

```
========================
Matching Evaluation:
========================

Robert:
Human best texture match in system worst! 31 27
Human best color match in system worst! 32 31
Human best texture match in system worst! 32 27
Human worst texture match in system best! 40 31
1: BEST COLOR MATCH RESULTS
TP: 22, FP: 0, FN: 18
Precision : 1.000000
Recall    : 0.550000
F1        : 0.709677
2: WORST COLOR MATCH RESULTS
TP: 17, FP: 1, FN: 23
Precision : 0.944444
Recall    : 0.425000
F1        : 0.586207
3: BEST TEXTURE MATCH RESULTS
TP: 12, FP: 1, FN: 28
Precision : 0.923077
Recall    : 0.300000
F1        : 0.452830
4: WORST TEXTURE MATCH RESULTS
TP: 17, FP: 2, FN: 23
Precision : 0.894737
Recall    : 0.425000
F1        : 0.576271
```

```
5: OVERALL COLOR AND TEXTURE MATCH RESULTS
TP: 68, FP: 4, FN: 92
Precision : 0.944444
Recall    : 0.425000
F1        : 0.586207

Jacky:
Human worst texture match in system best! 22 37
Human worst color match in system best! 31 6
Human worst texture match in system best! 31 40
Human worst texture match in system best! 38 19
1: BEST COLOR MATCH RESULTS
TP: 11, FP: 1, FN: 29
Precision : 0.916667
Recall    : 0.275000
F1        : 0.423077
2: WORST COLOR MATCH RESULTS
TP: 6, FP: 0, FN: 34
Precision : 1.000000
Recall    : 0.150000
F1        : 0.260870
3: BEST TEXTURE MATCH RESULTS
TP: 8, FP: 3, FN: 32
Precision : 0.727273
Recall    : 0.200000
F1        : 0.313725
4: WORST TEXTURE MATCH RESULTS
TP: 8, FP: 0, FN: 32
Precision : 1.000000
Recall    : 0.200000
F1        : 0.333333
5: OVERALL COLOR AND TEXTURE MATCH RESULTS
TP: 33, FP: 4, FN: 127
Precision : 0.891892
Recall    : 0.206250
F1        : 0.335025

Alex:
Human best color match in system worst! 31 22
Human best color match in system worst! 32 31
Human best texture match in system worst! 32 27
Human worst texture match in system best! 40 31
1: BEST COLOR MATCH RESULTS
TP: 20, FP: 0, FN: 20
Precision : 1.000000
Recall    : 0.500000
F1        : 0.666667
2: WORST COLOR MATCH RESULTS
TP: 10, FP: 2, FN: 30
Precision : 0.833333
Recall    : 0.250000
F1        : 0.384615
3: BEST TEXTURE MATCH RESULTS
TP: 10, FP: 1, FN: 30
Precision : 0.909091
```

```
Recall    : 0.250000
F1        : 0.392157
4: WORST TEXTURE MATCH RESULTS
TP: 13, FP: 1, FN: 27
Precision : 0.928571
Recall    : 0.325000
F1        : 0.481481
5: OVERALL COLOR AND TEXTURE MATCH RESULTS
TP: 53, FP: 4, FN: 107
Precision : 0.929825
Recall    : 0.331250
F1        : 0.488479

Ashley:
Human best texture match in system worst! 17 29
Human best texture match in system worst! 23 25
Human best texture match in system worst! 31 27
1: BEST COLOR MATCH RESULTS
TP: 18, FP: 0, FN: 22
Precision : 1.000000
Recall    : 0.450000
F1        : 0.620690
2: WORST COLOR MATCH RESULTS
TP: 9, FP: 0, FN: 31
Precision : 1.000000
Recall    : 0.225000
F1        : 0.367347
3: BEST TEXTURE MATCH RESULTS
TP: 9, FP: 0, FN: 31
Precision : 1.000000
Recall    : 0.225000
F1        : 0.367347
4: WORST TEXTURE MATCH RESULTS
TP: 7, FP: 3, FN: 33
Precision : 0.700000
Recall    : 0.175000
F1        : 0.280000
5: OVERALL COLOR AND TEXTURE MATCH RESULTS
TP: 43, FP: 3, FN: 117
Precision : 0.934783
Recall    : 0.268750
F1        : 0.417476


========================
Clustering Evaluation:
========================


Using r=0.2:

Robert:
TP: 55, FP: 66, TN: 551, FN: 108
Rand index: 0.776923
Precision : 0.454545
```

```
Recall    : 0.337423
F1        : 0.387324

Jacky:
TP: 60, FP: 68, TN: 549, FN: 103
Rand index: 0.780769
Precision : 0.468750
Recall    : 0.368098
F1        : 0.412371

Alex:
TP: 60, FP: 60, TN: 557, FN: 103
Rand index: 0.791026
Precision : 0.500000
Recall    : 0.368098
F1        : 0.424028

Ashley:
TP: 52, FP: 65, TN: 552, FN: 111
Rand index: 0.774359
Precision : 0.444444
Recall    : 0.319018
F1        : 0.371429


Using r=0.5:

Robert:
TP: 48, FP: 73, TN: 575, FN: 84
Rand index: 0.798718
Precision : 0.396694
Recall    : 0.363636
F1        : 0.379447

Jacky:
TP: 52, FP: 76, TN: 572, FN: 80
Rand index: 0.800000
Precision : 0.406250
Recall    : 0.393939
F1        : 0.400000

Alex:
TP: 41, FP: 79, TN: 569, FN: 91
Rand index: 0.782051
Precision : 0.341667
Recall    : 0.310606
F1        : 0.325397

Ashley:
TP: 35, FP: 82, TN: 566, FN: 97
Rand index: 0.770513
Precision : 0.299145
Recall    : 0.265152
F1        : 0.281124
```

```
Using r=0.8:

Robert:
TP: 44, FP: 77, TN: 542, FN: 117
Rand index: 0.751282
Precision : 0.363636
Recall    : 0.273292
F1        : 0.312057

Jacky:
TP: 47, FP: 81, TN: 538, FN: 114
Rand index: 0.750000
Precision : 0.367188
Recall    : 0.291925
F1        : 0.325260

Alex:
TP: 40, FP: 80, TN: 539, FN: 121
Rand index: 0.742308
Precision : 0.333333
Recall    : 0.248447
F1        : 0.284698

Ashley:
TP: 29, FP: 88, TN: 531, FN: 132
Rand index: 0.717949
Precision : 0.247863
Recall    : 0.180124
F1        : 0.208633
```

Code Listing

visretrieval.py

```
import os
import sys
import time
import cv2
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import gridspec as gridspec
from scipy.misc import comb
# from PIL import Image
# from numpy import linalg as la


# ============================================================
# Constants
# ============================================================

NUM_IM = 40
NUM_CLUSTERS = 7
COL_RANGE = 256
BINS = 8
BIN_SIZE = int(COL_RANGE/BINS)
BLK_THRESH = 40
IMG_H = 60
IMG_W = 89
LAP_BINS = 128
LAP_BIN_SZ = int(COL_RANGE*8)/LAP_BINS

# ============================================================
# Gross Color Matching
# ============================================================

def hexencode(rgb, factor):
    """Convert RGB tuple to hexadecimal color code."""
    r = rgb[0]*factor
    g = rgb[1]*factor
    b = rgb[2]*factor
    return '#%02x%02x%02x' % (r,g,b)

def visualize_chist(image, hist, colors, title):
    colors = sorted(colors, key=lambda c: -hist[(c[0])][(c[1])][(c[2])])
    plt.rcParams['font.family']='Aller Light'
    for idx, c in enumerate(colors):
        r = c[0]
        g = c[1]
        b = c[2]
        # print 'color, count:', hexencode(c, BIN_SIZE), hist[r][g][b]
        plt.subplot(1,2,1).bar(idx, hist[r][g][b], color=hexencode(c, BIN_SIZE),
edgecolor=hexencode(c, BIN_SIZE))
```

```
        plt.xticks([])
        # plt.subplot(1,2,2),plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
        # plt.xticks([]),plt.yticks([])
    dir_name = './color_hist/'
    if not os.path.exists(dir_name):
        os.makedirs(dir_name)
    title = dir_name+title+'.png'
    plt.savefig(title, bbox_inches='tight')
    plt.clf()
    plt.close('all')
    print title
    plot = cv2.imread(title, cv2.IMREAD_UNCHANGED)
    return plot
    # show(plot, 100)
    # clear image

def color_histogram(image, title):
    '''
    Calculate the 3D color histogram of an image by counting the number
    of RGB values in a set number of bins
    image -- pre-loaded image using cv2.imread function
    title -- image title
    (optional: visualize the histogram as a bar graph)
    '''
    colors = []
    h = len(image)
    w = len(image[0])
    # Create a 3D array - if BINS is 8, there are 8^3 = 512 total bins
    hist = np.zeros(shape=(BINS, BINS, BINS))
    # Traverse each pixel in the image matrix and increment the appropriate
    # hist[r_bin][g_bin][b_bin] - we know which one by floor dividing the
    # original RGB values / BIN_SIZE
    for i in xrange(h):
        for j in xrange(w):
            pixel = image[i][j]
            # If the pixel is below black threshold, do not count it
            if pixel[0] > BLK_THRESH and pixel[1] > BLK_THRESH \
            and pixel[2] > BLK_THRESH:
                # Note: pixel[i] is descending since OpenCV loads BGR
                r_bin = pixel[2] / BIN_SIZE
                g_bin = pixel[1] / BIN_SIZE
                b_bin = pixel[0] / BIN_SIZE
                hist[r_bin][g_bin][b_bin] += 1
                # Generate list of color keys for visualization
                if (r_bin,g_bin,b_bin) not in colors:
                    colors.append( (r_bin,g_bin,b_bin) )
    # plot = visualize_chist(image, hist, colors, title)
    # return plot, hist
    return hist

def l1_color_norm(h1, h2):
    diff = 0
    total = 0
    for r in xrange(0, BINS):
        for g in xrange(0, BINS):
```

```
            for b in range(0, BINS):
                diff += abs(h1[r][g][b] - h2[r][g][b])
                total += h1[r][g][b] + h2[r][g][b]
    l1_norm = diff / 2.0 / total
    similarity = 1 - l1_norm
    # print 'diff, sum and distance:', diff, sum, distance
    return l1_norm

def calc_cdistance(chists):
    chist_dis = {}
    for i in xrange(NUM_IM):
        for j in xrange(i, NUM_IM):
            if (i,j) not in chist_dis:
                d = l1_color_norm(chists[i], chists[j])
                chist_dis[(i,j)] = d
    # print chist_dis
    return chist_dis

def color_matches(k, chist_dis):
    """
    Find images most like and unlike an image based on color distribution.
    k -- the original image for comparison
    chists -- the list of color histograms for analysis
    """
    results = {}
    indices = []
    distances = []

    for i in xrange(0, NUM_IM):
        if k > i: # because tuples always begin with lower index
            results[i] = chist_dis[(i,k)]
        else:
            results[i] = chist_dis[(k,i)]
    # Ordered list of tuples (dist, idx) from most to least similar
    # -- first value will be the original image with diff of 0
    results = sorted([(v, k) for (k, v) in results.items()])
    # print 'results for image', k, results
    seven = results[:4]
    seven.extend(results[-3:])
    # print 'last seven for image', k, seven
    distances, indices = zip(*seven)
    # print 'distances:',distances
    # print 'indices:',indices
    return indices, distances

def find_four(chist_dis):
    results = {}
    # ensure that a<b, b<c and c<d as order does not matter
    for a in xrange(NUM_IM):
        for b in xrange(a+1,NUM_IM):
            for c in xrange(b+1,NUM_IM):
                for d in xrange(c+1,NUM_IM):
                    results[(a,b,c,d)] = \
                    chist_dis[(a,b)] + chist_dis[(a,c)] + \
                    chist_dis[(a,d)] + chist_dis[(b,c)] + \
```

```
                    chist_dis[(b,d)] + chist_dis[(c,d)]
    results = sorted([(v, k) for (k, v) in results.items()])
    best = results[0]
    worst = results[-1]
    indices = list(best[1])
    indices.extend(list(worst[1]))
    # print "results: ", len(results), #results
    # print "best, worst", best, worst
    return indices

# =============================================================
# Gross Texture Matching
# =============================================================

def grayscale(image, title):
    """Convert image into black and white with (R+G+B)/3"""
    gray = np.zeros(shape=(IMG_H, IMG_W))
    for row in xrange(IMG_H):
        for col in xrange(IMG_W):
            # compose gray image pixel by pixel
            pixel = image[row][col]
            rgb = int(pixel[0]) + int(pixel[1]) + int(pixel[2])
            g = int(round((rgb / 3.0),0)) # round up
            gray[row][col] = g
    # gray_img = cv2.imread(gray, cv2.IMREAD_UNCHANGED)
    cv2.imwrite('./gray/'+title+'.png', gray)
    return gray

def laplacian(gray, title):
    """Make laplacian image by calculating sum of neighbors for every pixel"""
    # gray = grayscale(image)
    laplacian = np.zeros(shape=(IMG_H, IMG_W))
    # laplacian.fill(COL_RANGE) # white would be 255+255+255/3 = 255
    for i in xrange(0,IMG_H):
        for j in xrange(0,IMG_W):
            # if gray[i][j] > int(BLK_THRESH/2):
            if gray[i][j]:
                neighbors = []
                for row in xrange(i-1,i+2):
                    for col in xrange(j-1,j+2):
                        if not (row is i and col is j) and index_valid(row,col):
                            neighbors.append(gray[row][col])
                neighbor_sum = reduce(lambda x, y: x+y, neighbors)
                n = len(neighbors)
                laplacian[i][j] = gray[i][j]*n - neighbor_sum
                # print 'calculations per pixel:', laplacian[i][j], gray[i][j], n,
neighbor_sum
    # print 'laplacian',laplacian
    cv2.imwrite('./laplacian/'+title+'.png', laplacian)
    return laplacian

def index_valid(row,col):
    if (row < 0 or col < 0 or row >= IMG_H or col >= IMG_W):
        return False
    return True
```

```
def visualize_thist(image, hist, bins, title):
    # bins = sorted(bins, key=lambda c: -hist[c])
    plt.rcParams['font.family']='Aller Light'
    for idx, c in enumerate(bins):
        plt.subplot(1,2,1).bar(c, hist[c], color='#D3D3D3')
    dir_name = './texture_hist/'
    if not os.path.exists(dir_name):
        os.makedirs(dir_name)
    title = dir_name+title+'.png'
    plt.savefig(title, bbox_inches='tight')
    plt.clf()
    plt.close('all')
    print title
    plot = cv2.imread(title, cv2.IMREAD_UNCHANGED)
    return plot
    # show(plot, 100)
    # clear image

def texture_histogram(lap, gray, title):
    hist = np.zeros(shape=LAP_BINS)
    bins = []
    for i in range(IMG_H):
        for j in range(IMG_W):
            pixel = lap[i][j]
            # Only count pixels outside the black threshold
            # to avoid image similarity due to shared black background
            if gray[i][j] > BLK_THRESH:
                if pixel < 256:
                    bin = abs(pixel/LAP_BIN_SZ)
                    hist[bin] += 1
                    if bin not in bins:
                        bins.append(bin)
    # plot = visualize_thist(gray, hist, bins, title)
    # print 'hist', hist
    # return hist, plot
    return hist

def l1_texture_norm(h1, h2):
    diff = 0
    total = 0
    for bin in xrange(0,LAP_BINS):
        diff += abs(h1[bin] - h2[bin])
        total += h1[bin] + h2[bin]
    l1_norm = diff / 2.0 / total
    similarity = 1 - l1_norm
    return l1_norm

def calc_tdistance(thists):
    thist_dis = {}
    for i in xrange(NUM_IM):
        for j in xrange(i, NUM_IM):
            if (i,j) not in thist_dis:
                d = l1_texture_norm(thists[i], thists[j])
                thist_dis[(i,j)] = d
```

```
    return thist_dis

def texture_matches(k, thist_dis):
    results = {}
    indices = []
    distances = []
    for i in xrange(0, NUM_IM):
        if k > i: # because tuples always begin with lower index
            results[i] = thist_dis[(i,k)]
        else:
            results[i] = thist_dis[(k,i)]
    results = sorted([(v, k) for (k, v) in results.items()])
    seven = results[:4]
    seven.extend(results[-3:])
    distances, indices = zip(*seven)
    return indices, distances

# ============================================================
# Combine Similarities and Cluster
# ============================================================

def combine_similarities(chist_dis, thist_dis, r):
    '''where r is the ratio'''
    similarities = {}
    distances = {}
    closest = 1
    for i in xrange(NUM_IM):
        for j in xrange(i+1, NUM_IM):
            if (i,j) not in similarities:
                t_sim = 1 - thist_dis[(i,j)]
                c_sim = 1 - chist_dis[(i,j)]
                s = r*(t_sim) + (1-r)*(c_sim)
                d = 1 - s
                # d = r*thist_dis[(i,j)] + (1-r)*chist_dis[(i,j)]
                similarities[(i,j)] = s
                distances[(i,j)] = d
                # if (d < closest):
                #     closest = (i,j)
    return similarities, distances#, closest

# 0 is COMPLETE LINK, 1 is SINGLE
def cluster(distances, link):
    clusters = {}
    for idx in xrange(0,NUM_IM):
        clusters[(idx,)] = (idx,)
    # print clusters
    counter = 0
    while len(clusters) > NUM_CLUSTERS:
        nearest_pair = (None, None)
        nearest_dist = 1 # total dissimilarity
        counter += 1
        # Go through every cluster-pair to find nearest pair
        for a in clusters:
            for b in clusters:
                # Do not compare with same cluster
```

```
                if a is not b:
                    dist = link
                    # For each element in both clusters, determine "nearness"
                    # Complete nearness: farthest distance bt any 2 el in 2 clusters
                    # Single nearness: nearest distance bt any 2 el in 2 clusters
                    for i in clusters[a]:
                        for j in clusters[b]:
                            k = (i,j)
                            if i > j:
                                k = (j,i) # tuples always in ascending order
                            curr_dist = distances[k]
                            if (link is 0 and curr_dist > dist) or (link is 1 and
curr_dist < dist):
                                # print counter, ': New distance for', clusters[a],
clusters[b], dist, '->', curr_dist
                                dist = curr_dist
                    # Find out if this is the nearest pair so far in the iteration
                    if dist < nearest_dist:
                        # print counter, ': Replace distance with ', (a,b),
nearest_dist, '->', dist
                        nearest_dist = dist
                        nearest_pair = (a,b)
                        nearest_pair_values = clusters[a]+clusters[b] # add elements
in a tuple
        # combine the nearest pair of clusters and remove old clusters
        clusters.pop(nearest_pair[0])
        clusters.pop(nearest_pair[1])
        # new_pair = nearest_pair[0] + nearest_pair[1]
        clusters[nearest_pair_values] = nearest_pair_values
        # clusters[new_pair] = clusters[new_pair]
        # print counter, clusters.keys()

    # print clusters
    clusters = clusters.keys()
    return clusters

# ============================================================
# Performance Measure
# ============================================================

def loadCSV(filename):
    results = []
    infile = open(filename, 'rU')
    for idx in xrange(NUM_IM):
        results.append(idx)
        data = infile.readline().strip().replace(' ', '').split(',')
        # append img idx to results list, then append friend results
        # for best color, worst color, best texture, worst texture
        # results.extend(data)
        for j in xrange(4):
            results.append(int(data[j])-1)
    return results

def print_results(tp, fp, fn):
    print "TP: %d, FP: %d, FN: %d" % (tp, fp, fn)
```

```
    precision = float(tp) / (tp + fp)
    recall = float(tp) / (tp + fn)

    print "Precision : %f" % precision
    print "Recall    : %f" % recall
    print "F1        : %f" % ((2.0 * precision * recall) / (precision + recall))

def match_eval(cresults, tresults, hresults):
    '''
    tp -- if human guess is in sys results
    fn -- if human guess is missing from sys results
    fp -- if human guess is in opposite of sys results
    '''
    # best,worst color and best,worst texture measures
    bc_tp, bc_fp, bc_fn = 0,0,0
    wc_tp, wc_fp, wc_fn = 0,0,0
    bt_tp, bt_fp, bt_fn = 0,0,0
    wt_tp, wt_fp, wt_fn = 0,0,0

    counter = 0

    for i in xrange(0,NUM_IM*7,7):

        # range indices for best/worst system results
        b_from = i+1
        b_to = i+4
        w_from = i+4
        w_to = i+7

        # human results for best color, worst color,
        # best texture, worst texture
        # note: need var j because human list increments by 5
        # whereas system list increments by 7
        j = i - (2*counter)
        c_best = hresults[j+1]
        c_worst = hresults[j+2]
        t_best = hresults[j+3]
        t_worst = hresults[j+4]
        counter += 1

        # check best color match
        if c_best in cresults[b_from:b_to]:
            bc_tp += 1
        else:
            bc_fn += 1
        # since we are measuring computer against human perf:
        # program messed up and made a big false positive
        if c_best in cresults[w_from:w_to]:
            wc_fp += 1
            print 'Human best color match in system worst!', cresults[i]+1, c_best+1
            # +1 to make it readable - display current index and image in q

        # check worst color match
        if c_worst in cresults[w_from:w_to]:
```

```
            wc_tp += 1
        else:
            wc_fn += 1
        if c_worst in cresults[b_from:b_to]:
            bc_fp += 1
            print'Human worst color match in system best!', cresults[i]+1, c_worst+1

        # check best texture match
        if t_best in tresults[b_from:b_to]:
            bt_tp += 1
        else:
            bt_fn += 1
        # since we are measuring computer against human perf:
        # program messed up and made a big false positive
        if t_best in tresults[w_from:w_to]:
            wt_fp += 1
            print 'Human best texture match in system worst!', cresults[i]+1,t_best+1

        # check worst texture match
        if t_worst in tresults[w_from:w_to]:
            wt_tp += 1
        else:
            wt_fn += 1
        if t_worst in tresults[b_from:b_to]:
            bt_fp += 1
            print'Human worst texture match in system best!', cresults[i]+1,t_worst+1

    # Calculate total tp,fp,fn values
    tp = bc_tp + wc_tp + bt_tp + wt_tp
    fp = bc_fp + wc_fp + bt_fp + wt_fp
    fn = bc_fn + wc_fn + bt_fn + wt_fn

    print "1: BEST COLOR MATCH RESULTS"
    print_results(bc_tp, bc_fp, bc_fn)

    print "2: WORST COLOR MATCH RESULTS"
    print_results(wc_tp, wc_fp, wc_fn)

    print "3: BEST TEXTURE MATCH RESULTS"
    print_results(bt_tp, bt_fp, bt_fn)

    print "4: WORST TEXTURE MATCH RESULTS"
    print_results(wt_tp, wt_fp, wt_fn)

    print "5: OVERALL COLOR AND TEXTURE MATCH RESULTS"
    print_results(tp, fp, fn)

def cluster_id(list_of_tups, key):
    '''
    Take list of cluster tups and return list of cluster id's
    where each list index is the image and the el is its set id
    key -- 1 if its starting index needs to be decremented to 0
    '''
    cluster_ids = [0] * 40
    for i in xrange(7):
```

```
        tup = list_of_tups[i]
        for j in tup:
            if key is 1:
                cluster_ids[j-1] = i
            else:
                cluster_ids[j] = i
    return cluster_ids

def cluster_eval(system, human):
    # Change list of tuples -> list of lists
    hum = cluster_id(human, 1)
    syst = cluster_id(system, 0)

    # Initialize True-Pos, True-Neg False-Pos, False-Neg measures
    tp, tn, fp, fn = 0,0,0,0
    for i in xrange(NUM_IM):
        for j in xrange(i+1,NUM_IM):
            if hum[i] == hum[j] and syst[i] == syst[j]:
                tp += 1
            if hum[i] == hum[j] and syst[i] != syst[j]:
                fp += 1
            if hum[i] != hum[j] and syst[i] == syst[j]:
                fn += 1
            if hum[i] != hum[j] and syst[i] != syst[j]:
                tn += 1
    print "TP: %d, FP: %d, TN: %d, FN: %d" % (tp, fp, tn, fn)

    rand_idx = (float(tp + tn) / (tp + fp + fn + tn))
    precision = float(tp) / (tp + fp)
    recall = float(tp) / (tp + fn)

    # Print results:
    print "Rand index: %f" % rand_idx
    print "Precision : %f" % precision
    print "Recall    : %f" % recall
    print "F1        : %f" % ((2.0 * precision * recall) / (precision + recall))

    return rand_idx

# ============================================================
# Helper and Display Functions
# ============================================================

def save(image, name):
    cv2.imwrite(name, image)

def show(image, wait):
    cv2.waitKey(wait)
    cv2.imshow('Image', image)

def display_all(images, titles):
    plt.rcParams['font.family']='Aller Light'
    for i in xrange(NUM_IM):
        plt.subplot(5,8,i+1),plt.imshow(cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB)) #
row, col
```

```
        plt.title(titles[i], size=12)
        plt.xticks([]),plt.yticks([])
    title = './all_im.png'
    plt.savefig(title, bbox_inches='tight')
    print title

def pair_stitch_v(images1, images2, titles):
    # note: must be same width
    n = len(images1)
    for i in xrange(0, n):
        img = np.concatenate((images1[i], images2[i]), axis=0)
        path = './'+titles[i]+'.png'
        cv2.imwrite(path, img)

def septuple_stitch_h(images, titles, dir_name, cresults, cdistances, cvt):
    plt.rcParams['font.family']='Aller Light'
    gs1 = gridspec.GridSpec(1,7)
    gs1.update(wspace=0.05, hspace=0.05) # set the spacing between axes.
    for k in xrange(0, NUM_IM*7, 7):
        for i in xrange(7):
            idx = cresults[k+i]
            ax = plt.subplot(gs1[i])
            plt.axis('on')
            if cvt is 0:
                plt.imshow(images[idx], cmap="Greys_r")
            elif cvt is -1:
                plt.imshow(images[idx], cmap="binary")
            else:
                plt.imshow(cv2.cvtColor(images[idx], cv2.COLOR_BGR2RGB)) # row, col
            plt.xticks([]),plt.yticks([])
            if cdistances:
                if i == 0:
                    plt.xlabel('similarity:')
                else:
                    sim = 1 - round(cdistances[k+i], 5)
                    plt.xlabel(sim)
                plt.title(titles[idx], size=12)
            ax.set_aspect('equal')

        title = titles[k/7]
        if not os.path.exists(dir_name):
            os.makedirs(dir_name)
        title = dir_name+title+'.png'
        plt.savefig(title, bbox_inches='tight')
        print title
        plt.clf()
        plt.close('all')

def four_stitch_h(images, titles, cresults, dir_name):
    plt.rcParams['font.family']='Aller Light'
    gs1 = gridspec.GridSpec(1,4)
    gs1.update(wspace=0.05, hspace=0.05) # set the spacing between axes.
    for k in xrange(0,8,4):
        for i in xrange(4):
            idx = cresults[i+k]
```

```
            ax = plt.subplot(gs1[i])
            plt.axis('on')
            plt.imshow(cv2.cvtColor(images[idx], cv2.COLOR_BGR2RGB)) # row, col
            plt.xticks([]),plt.yticks([])
            plt.title(titles[idx], size=12)
            ax.set_aspect('equal')
        if k is 0:
            title = 'best_match.png'
        else:
            title = 'worst_match.png'
        if not os.path.exists(dir_name):
            os.makedirs(dir_name)
        plt.savefig(dir_name+title, bbox_inches='tight')
        print title
        plt.clf()
        plt.close('all')

def cluster_stitch_h(images, titles, clusters, link, dir_name):
    # set n to the length of the largest cluster
    n = 1
    for cluster in clusters:
        if len(cluster) > n:
            n = len(cluster)
    plt.rcParams['font.family']='Aller Light'
    gs1 = gridspec.GridSpec(7,n)
    gs1.update(wspace=0.05, hspace=0.05) # set the spacing between axes.
    for k in xrange(0, n*7, n):
        cluster = clusters[(k/n)]
        for i in xrange(len(cluster)):
            idx = cluster[i]
            ax = plt.subplot(gs1[i+k])
            plt.axis('on')
            plt.imshow(cv2.cvtColor(images[idx], cv2.COLOR_BGR2RGB)) # row, col
            plt.xticks([]),plt.yticks([])
            plt.title(titles[idx], size=12)
            ax.set_aspect('equal')
    if link is 0:
        title = 'cluster_complete'
    else:
        title = 'cluster_single'
    if not os.path.exists(dir_name):
        os.makedirs(dir_name)
    title = dir_name+title+'.png'
    plt.savefig(title, bbox_inches='tight')
    print title
    plt.clf()
    plt.close('all')

# ============================================================
# Where All the Magic Gets Invoked
# ============================================================

def main():

    # Check if user has provided a directory argument
```

```
    if len(sys.argv) < 2:
        sys.exit("Need to specify a path from which to read images")

    format = ".ppm"
    path = "./" + sys.argv[1]

    # ================================
    # Parts 1 and 2: gross color and texture matching
    # ================================

    # 1: color
    images = []
    titles = []
    chists = []
    chist_images = []
    cresults = []
    cdistances = []

    # 2: texture
    gray_images = []
    lap_images = []
    thists = []
    thist_images = []
    tresults = []
    tdistances = []

    # Process images from user-provided directory
    if os.path.exists(path):
        imfilelist=[os.path.join(path,f) for f in os.listdir(path) if
f.endswith(format)]
        if len(imfilelist) < 1:
            sys.exit ("Need to specify a path containing .ppm files")
        NUM_IM = len(imfilelist) # default is 40
        for el in imfilelist:
            print(el)
            # Update images and titles list
            image = cv2.imread(el, cv2.IMREAD_UNCHANGED)
            title = el[9:-4]
            images.append(image)
            titles.append(title)
            # Generate color histogram
            chist = color_histogram(image, title)
            chists.append(chist)
            # chist_images.append(plot)
            # Generate texture histogram
            gray = grayscale(image, title)
            lap = laplacian(gray, title)
            # thist = texture_histogram(lap, gray, title)
            thist = texture_histogram(lap, gray, title)
            # thist,plot = texture_histogram(lap, gray, title)
            gray_images.append(gray)
            lap_images.append(lap)
            thists.append(thist)
            # thist_images.append(plot)
    else:
```

```
        sys.exit("The path name does not exist")

    # Calculate lookup table for distances based on color histograms
    chist_dis = calc_cdistance(chists)
    thist_dis = calc_tdistance(thists)

    # Determine 3 closest and 3 farthest matches for all images
    for k in xrange(NUM_IM):
        # By color
        results, distances = color_matches(k, chist_dis)
        cresults.extend(results)
        cdistances.extend(distances)
        # By texture
        res, dis = texture_matches(k, thist_dis)
        tresults.extend(res)
        tdistances.extend(dis)

    # Find set of 4 most different and 4 most similar images
    cfour = find_four(chist_dis)
    tfour = find_four(thist_dis)

    # ==================================
    # Note: Commented out image saving and display
    # ==================================

    # Display all images
    # display_all(images,titles)

    # Display septuples
   # septuple_stitch_h(images, titles, './part1/color_sim/', cresults, cdistances, 1)
    # septuple_stitch_h(chist_images, titles, './part1/color_hist_sim_untitled/',
cresults, None, 1)
    # for i in xrange(NUM_IM):
    #     pic_stitch(cresults[i], images, titles)
    # septuple_stitch_h(images, titles, './part2/tpics/', tresults, tdistances, 1)
 # septuple_stitch_h(thist_images, titles, './part2/thistograms/', tresults, None, 0)
 # septuple_stitch_h(gray_images, titles, './part2/gray_images/', tresults, None, 0)
 # septuple_stitch_h(lap_images, titles, './part2/lap_images/', tresults, None, -1)

    # Display four best and four worst, by color and by texture
    # four_stitch_h(images, titles, cfour, './part1/')
    # four_stitch_h(images, titles, tfour, './part2/')

    # ==================================
    # Part 3: combine similarities and cluster
    # ==================================

    similarities = []
    distances = []
    complete = []
    single = []

    # Testing combined similarities
    similarities, distances = combine_similarities(chist_dis, thist_dis, 0.2)
    combo_four = find_four(distances)
```

```
# four_stitch_h(images, titles, combo_four, './part3/')

complete = cluster(distances,0)
single = cluster(distances,1)
cluster_stitch_h(images, titles, complete, 0, './part3/')
cluster_stitch_h(images, titles, single, 1, './part3/')

# Testing additional r values for combined similarity to see their effect
sim, dis = combine_similarities(chist_dis, thist_dis, 0.5)
comp = cluster(dis,0)
sim2, dis2 = combine_similarities(chist_dis, thist_dis, 0.8)
comp2 = cluster(dis2,0)

# ================================
# Part 4: creative step
# ================================

robert = loadCSV('./part4/Robert.csv')
jacky = loadCSV('./part4/Jacky.csv')
alex = loadCSV('./part4/Alex.csv')
ashley = loadCSV('./part4/Ashley.csv')

# clusters
# TODO: load the CSV file instead of this manual stuff
jacky_c = \
[(2,17,23), \
(25,33,26,28), \
(18,19,20,21,22), \
(31,32,27), \
(13,30,29,34), \
(1,3,4,8,10,24,36,37,38,16), \
(5,6,7,9,11,12,14,15,35,39,40)]
robert_c = \
[(37,38,19,24), \
(2,13,11,10,16,7,5,6,15,9,14,12), \
(3,1,8,4), \
(27,31,32,), \
(29,34,17,30,36,39), \
(35,33,40,20,23), \
(25,26,28,18)]
alex_c = \
[(8,1,4,3,24,10,37,38,16,19),
(2,39,21,22),
(13,12,14),
(11,7,6,9),
(15,20,23,33,5,6),
(40,25,17,34,35,18,26,28),
(31,32,29,27,30)]
ashley_c = \
[(1,3,4,8,10,16),
(11,15,5,6,23,40,20,33,7),
(34,2),
(28,18,21,39,26,17,35,25),
(12,9,13,14),
(27,32,31),
```

```
    (37,19,22,36,29,24,38,30)]

    # print 'Robert', robert, robert_c
    # print 'Jacky', jacky, jacky_c
    # print 'Alex', alex, alex_c
    # print 'Ashley', ashley, ashley_c

    print '\n\n========================'
    print 'Matching Evaluation:'
    print '========================'
    print '\nRobert:'
    match_eval(cresults, tresults, robert)
    print '\nJacky:'
    match_eval(cresults, tresults, jacky)
    print '\nAlex:'
    match_eval(cresults, tresults, alex)
    print '\nAshley:'
    match_eval(cresults, tresults, ashley)

    print '\n\n========================'
    print 'Clustering Evaluation:'
    print '========================'

    print '\n\nUsing r=0.2:'
    print '\nRobert:'
    cluster_eval(complete, robert_c)
    print '\nJacky:'
    cluster_eval(complete, jacky_c)
    print '\nAlex:'
    cluster_eval(complete, alex_c)
    print '\nAshley:'
    cluster_eval(complete, ashley_c)

    print '\n\nUsing r=0.5:'
    print '\nRobert:'
    cluster_eval(comp, robert_c)
    print '\nJacky:'
    cluster_eval(comp, jacky_c)
    print '\nAlex:'
    cluster_eval(comp, alex_c)
    print '\nAshley:'
    cluster_eval(comp, ashley_c)

    print '\n\nUsing r=0.8:'
    print '\nRobert:'
    cluster_eval(comp2, robert_c)
    print '\nJacky:'
    cluster_eval(comp2, jacky_c)
    print '\nAlex:'
    cluster_eval(comp2, alex_c)
    print '\nAshley:'
    cluster_eval(comp2, ashley_c)

if __name__ == "__main__": main()
```

Appendices

Appendix A: User Studies

Ashley

Image Index	Color Best Match	Color Worst Match	Texture Best Match	Texture Worst Match
i01	10	6	10	26
i02	34	31	16	30
i03	4	7	8	30
i04	3	7	3	13
i05	6	16	16	7
i06	11	10	5	7
i07	9	10	40	37
i08	4	11	3	33
i09	40	38	12	30
i10	1	40	1	26
i11	7	38	6	1
i12	14	37	14	25
i13	39	25	12	34
i14	12	29	12	29
i15	40	1	14	30
i16	3	7	6	1
i17	20	29	29	13
i18	35	31	35	26
i19	36	26	15	02
i20	15	37	36	1
i21	39	31	39	24
i22	36	29	36	25
i23	5	24	24	1
i24	37	23	23	1
i25	26	10	26	37
i26	25	10	25	37

	Color		Texture	
127	32	35	27	37
128	25	27	35	31
129	22	23	36	1
130	29	11	29	39
131	32	30	27	39
132	27	18	31	39
133	7	38	34	26
134	2	32	33	39
135	40	24	36	1
136	22	6	22	25
137	24	18	24	13
138	36	11	10	30
139	21	11	21	1
140	7	4	7	1

Group into
7 Clusters:

11, 15,
5, 6,
23, 40
20, 33, 7
red

1, 3, 4,
8, 10, 16
Green

34, 02

12, 9
13, 14
Braun

yellow

37, 19, 22
36, 29
24, 38
30
green

28, 18
21, 39,
26, 17
35, 25
orange

27, 32
31
spiral

Alex

Image Index	Color		Texture	
	Best Match	Worst Match	Best Match	Worst Match
101	04	23	10	35
102	21	31	35	31
103	10	05	08	13
104	24	37	24	26
105	06	37	06	31
106	05	24	05	30
107	09	39	06	21
108	19	15	03	18
109	07	30	07	20
110	16	15	01	31
111	07	39	16	26
112	14	31	14	26
113	14	37	14	26
114	13	18	12	26
115	20	38	21	26
116	01	05	11	26
117	18	32	05	26
118	17	32	19	31
119	08	20	18	27
120	15	19	21	31
121	39	31	20	26
122	29	31	36	31
123	33	37	18	26
124	37	23	04	31
125	28	27	26	39
126	25	27	34	39

i27	32	26	32	39
i28	25	32	40	26
i29	22	31	30	31
i30	29	31	29	26
i31	22	29	32	29
i32	31	35	27	30
i33	23	37	34	27
i34	39	31	33	31
i35	17	32	36	31
i36	24	23	37	31
i37	24	06	36	26
i38	22	20	10	26
i39	34	31	38	26
i40	28	32	28	31

Group into
7 Clusters:

GREEN
i08 i01
i09 i03
i36 i24 i10
i37 i38
i16 i19

WHITE-YELLOW
i02
i39
i21
i22

BLAND
i13
i12
i14

BROWN
i11
i07
i06
i09

i40
i25
i17
i34
i35
i18
i26
i28
ORANGE

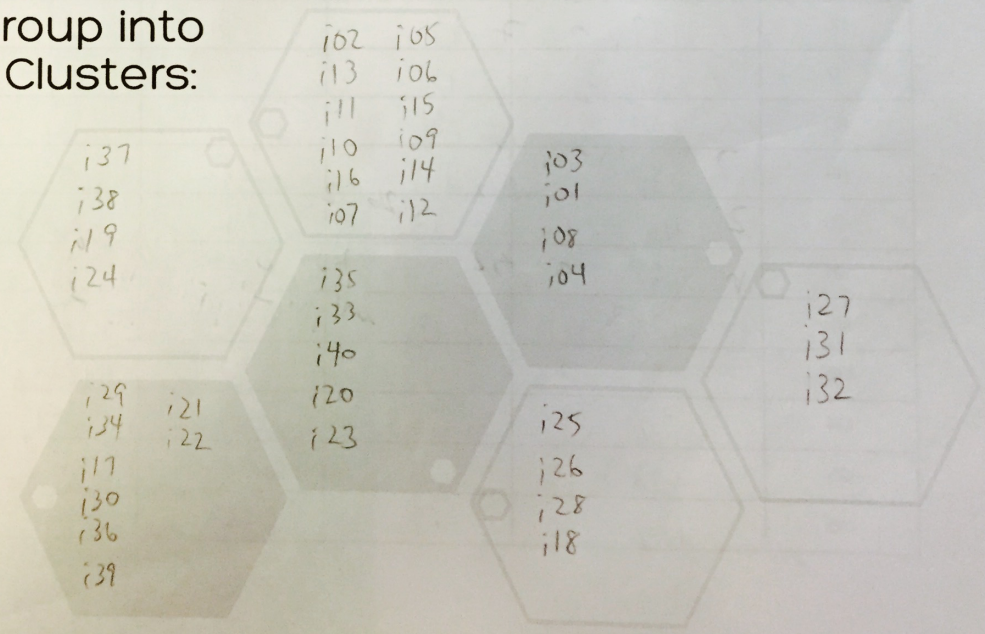
i15
i20
i23
i33
i05
i06
RED

i31
i32
i29
i27
i30
NEON

Image Index	Color		Texture	
	Best Match	Worst Match	Best Match	Worst Match
i01	i10	i31	i10	i26
i02	i34	i32	i16	i27
i03	i04	i15	i08	i11
i04	i03	i05	i03	i39
i05	i06	i27	i06	i26
i06	i05	i32	i05	i26
i07	i09	i02	i40	i29
i08	i24	i31	i13	i29
i09	i07	i04	i13	i29
i10	i01	i11	i01	i29
i11	i06	i32	i06	i29
i12	i39	i37	i14	i29
i13	i29	i31	i09	i26
i14	i39	i37	i12	i26
i15	i09	i16	i36	i29
i16	i37	i31	i31	i29
i17	i35	i31	i20	i11
i18	i35	i31	i19	i02
i19	i36	i31	i18	i26
i20	i15	i32	i17	i36
i21	i28	i32	i22	i11
i22	i29	i35	i21	i11
i23	i33	i37	i24	i26
i24	i36	i23	i23	i29
i25	i28	i31	i26	i08
i26	i17	i31	i25	i34

127	i30	i06	i31	i37
128	i25	i31	i24	i26
129	i22	i11	i30	i01
130	i27	i06	i29	i01
131	i32	i24	i27	i26
132	i31	i11	i27	i11
133	i40	i37	i28	i30
134	i02	i31	i21	i02
135	i18	i32	i36	i11
136	i24	i31	i35	i11
137	i16	i11	i36	i11
138	i36	i11	i10	i11
139	i34	i31	i13	i15
140	i35	i37	i07	i31

Group into
7 Clusters:

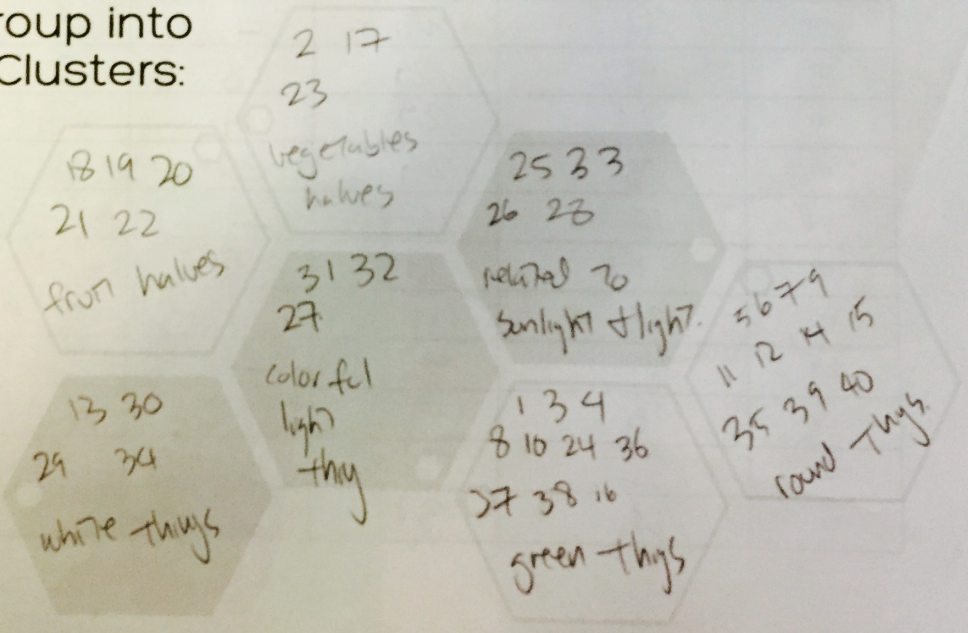


Jacky

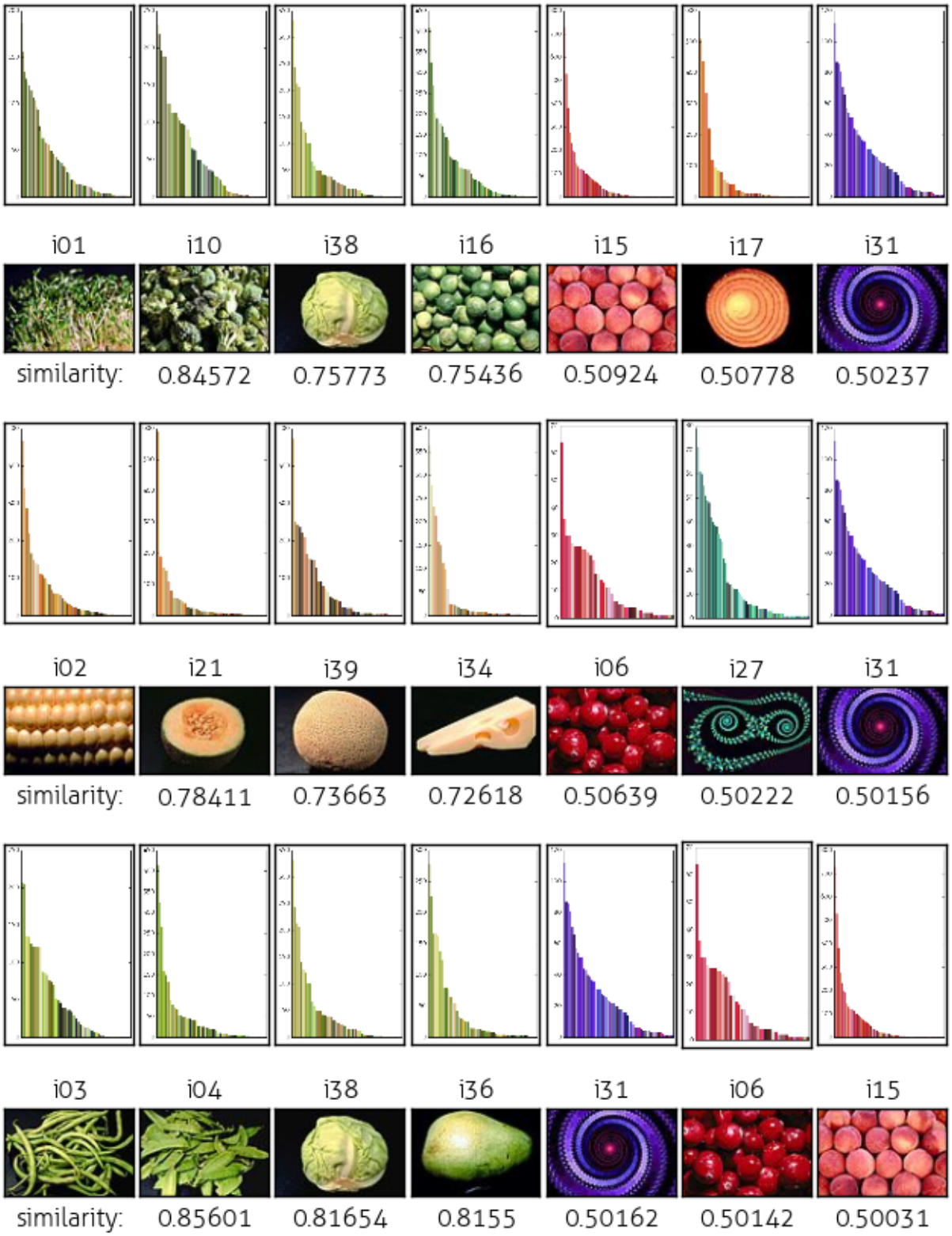
Image Index	Color		Texture	
	Best Match	Worst Match	Best Match	Worst Match
I01	10	5	10	18
I02	34	5	40	18
I03	4	5	4	20
I04	3	5	3	27
I05	6	29	6	26
I06	5	29 4	5	31
I07	40	29	40	28
I08	1	5	4	30
I09	40	29	14	29
I10	1	5	4	25
I11	6	29	5	33
I12	14	29	14	17
I13	39	29	14	32
I14	20	29	13	20
I15	12	29	11	31
I16	4	5	4	26
I17	18	29	18	2
I18	21	29	19	34
I19	3	29	20	34
I20	15	29	21	30
I21	18	6	22	27
I22	29	11	19	37
I23	5	31	24	34
I24	3	5	23	35
I25	26	31	26	21
I26	25	31	25	16

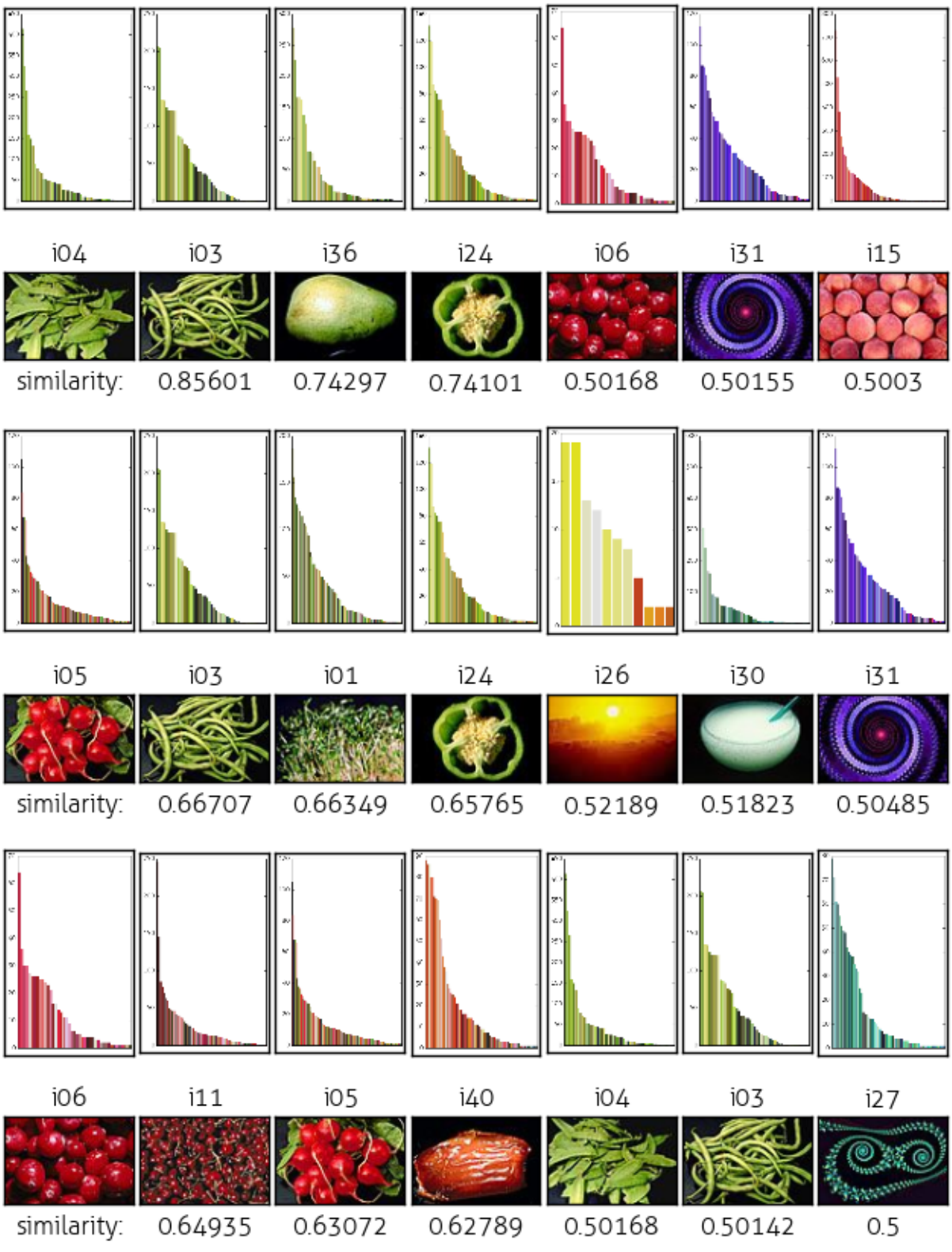
127	32	29	31	37
128	26	34	32	40
129	34	6	30	26
130	34	6	29	33
131	32	6	32	40
132	27	6	31	19
133	26	31	32	12
134	29	6	16	27
135	18	3	40	16
136	16	5	37	27
137	16	5	36	17
138	10	5	4	14
139	15	6	12	32
140	12	31	7	33

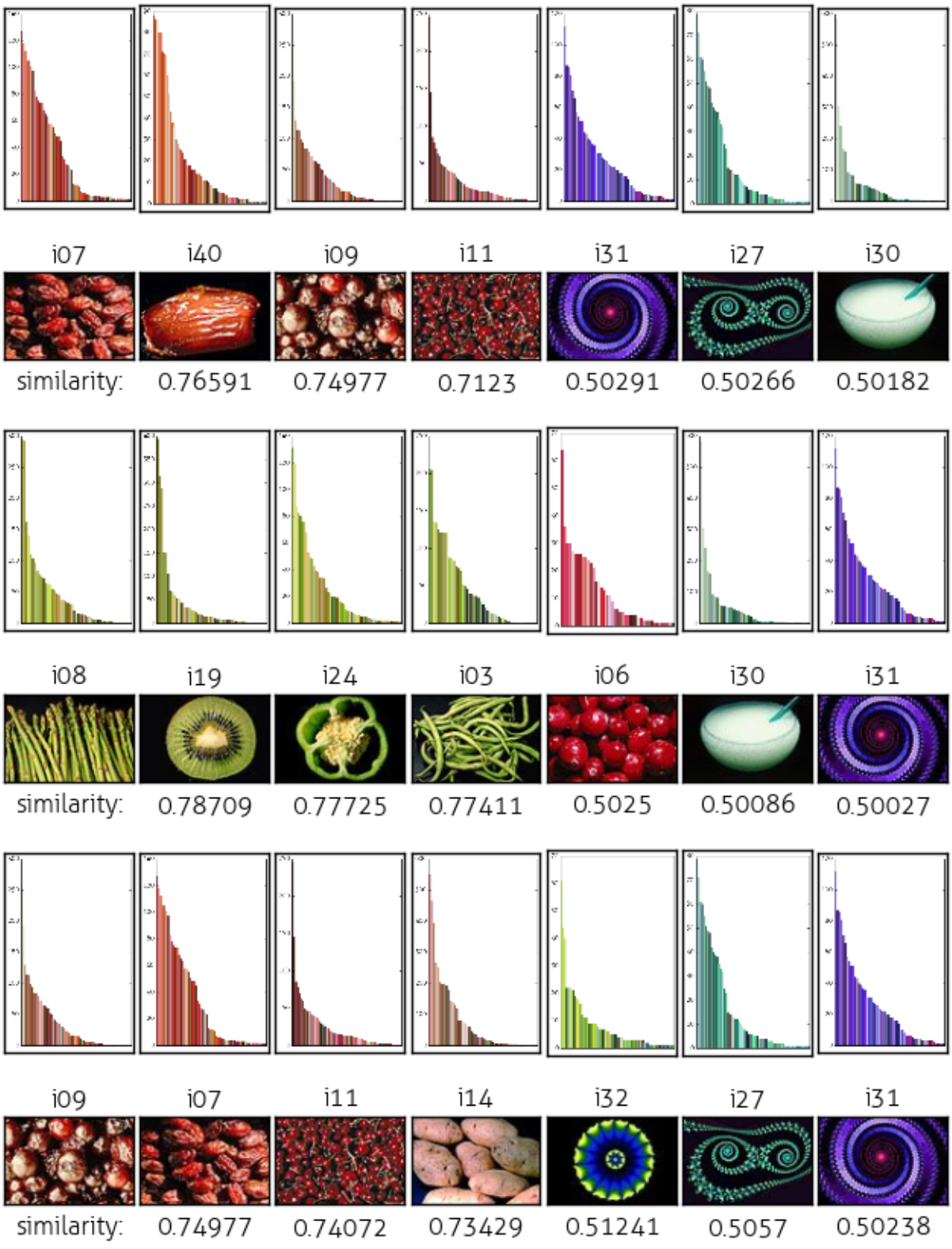
Group into
7 Clusters:

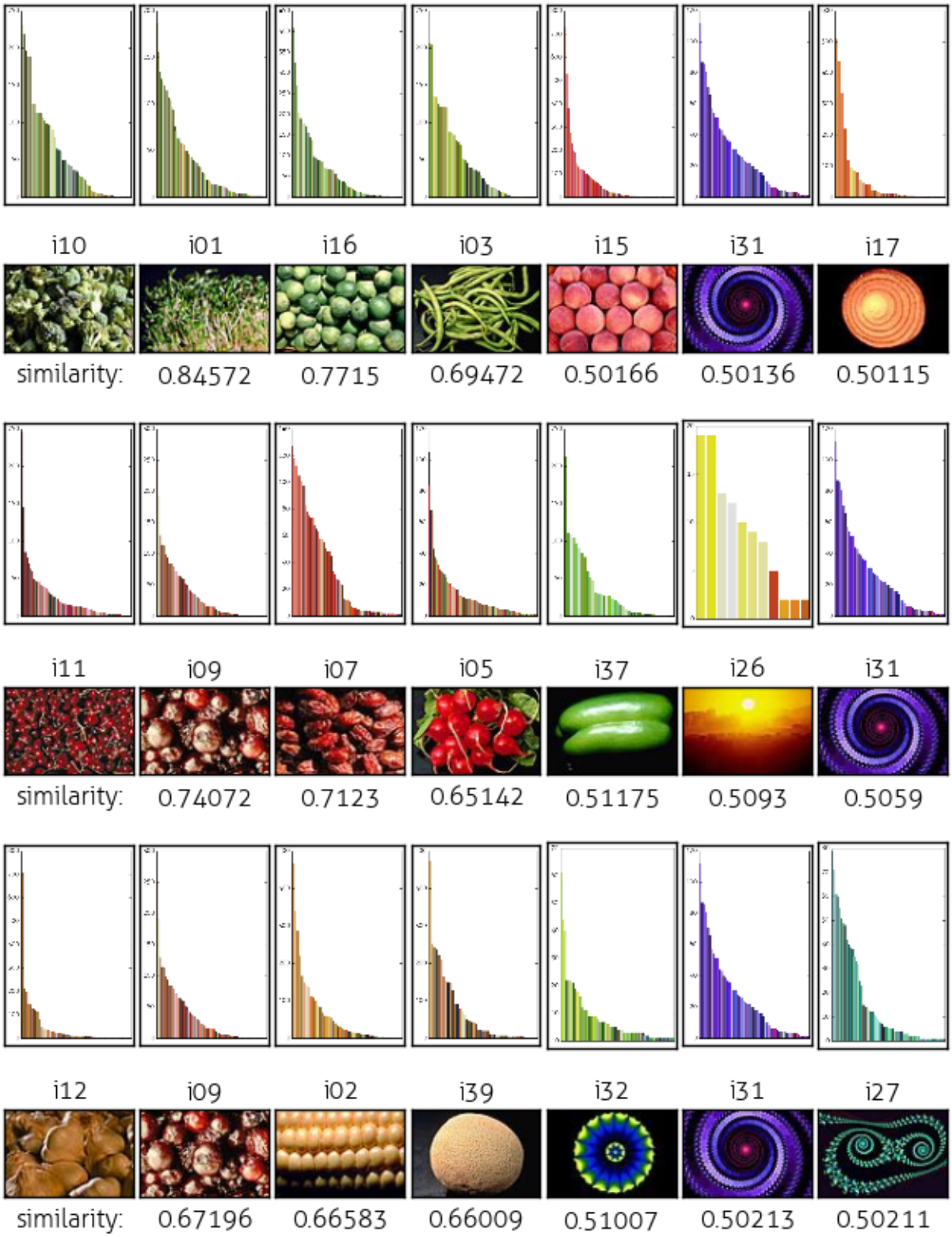


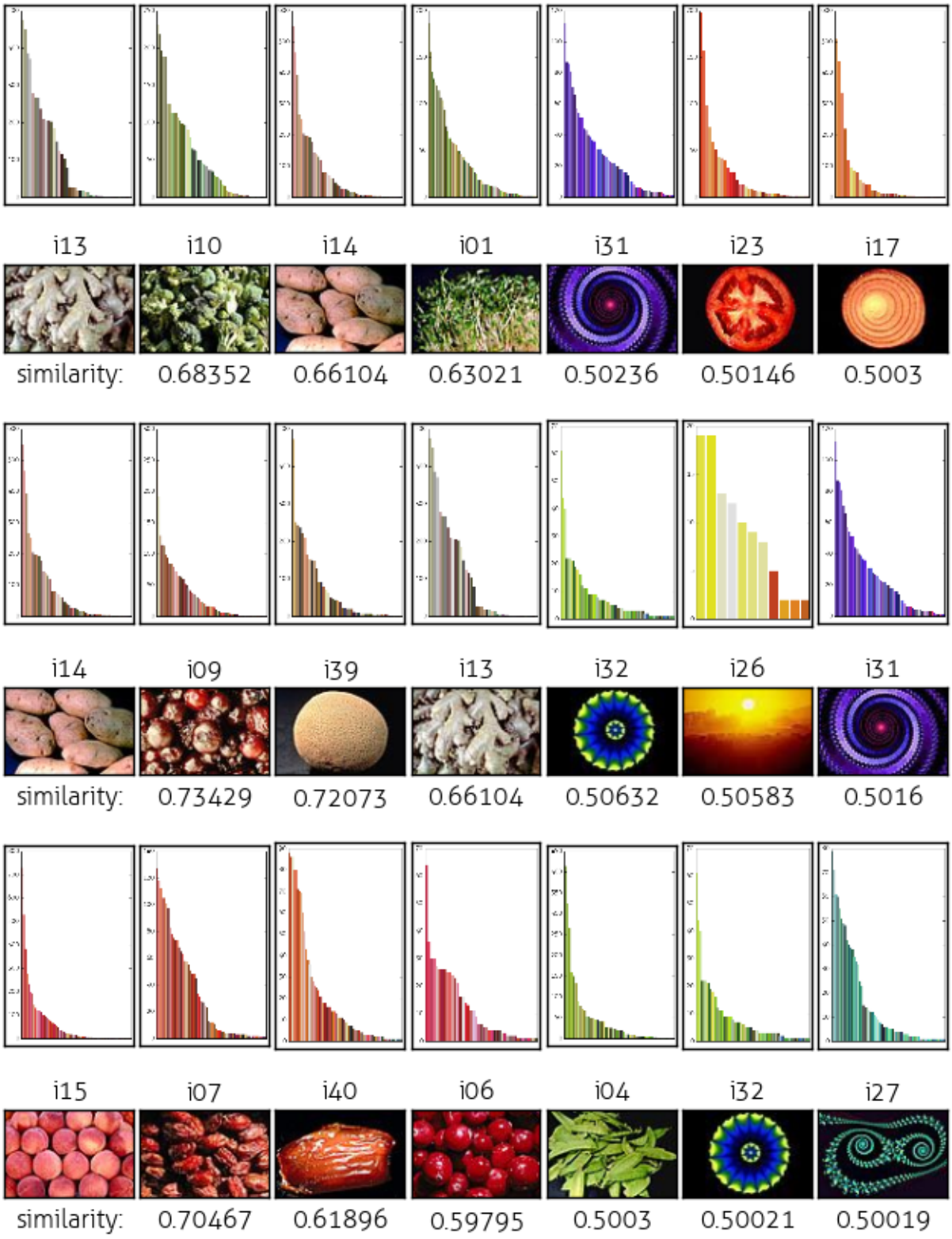
Appendix B: Septuples+Color Histograms

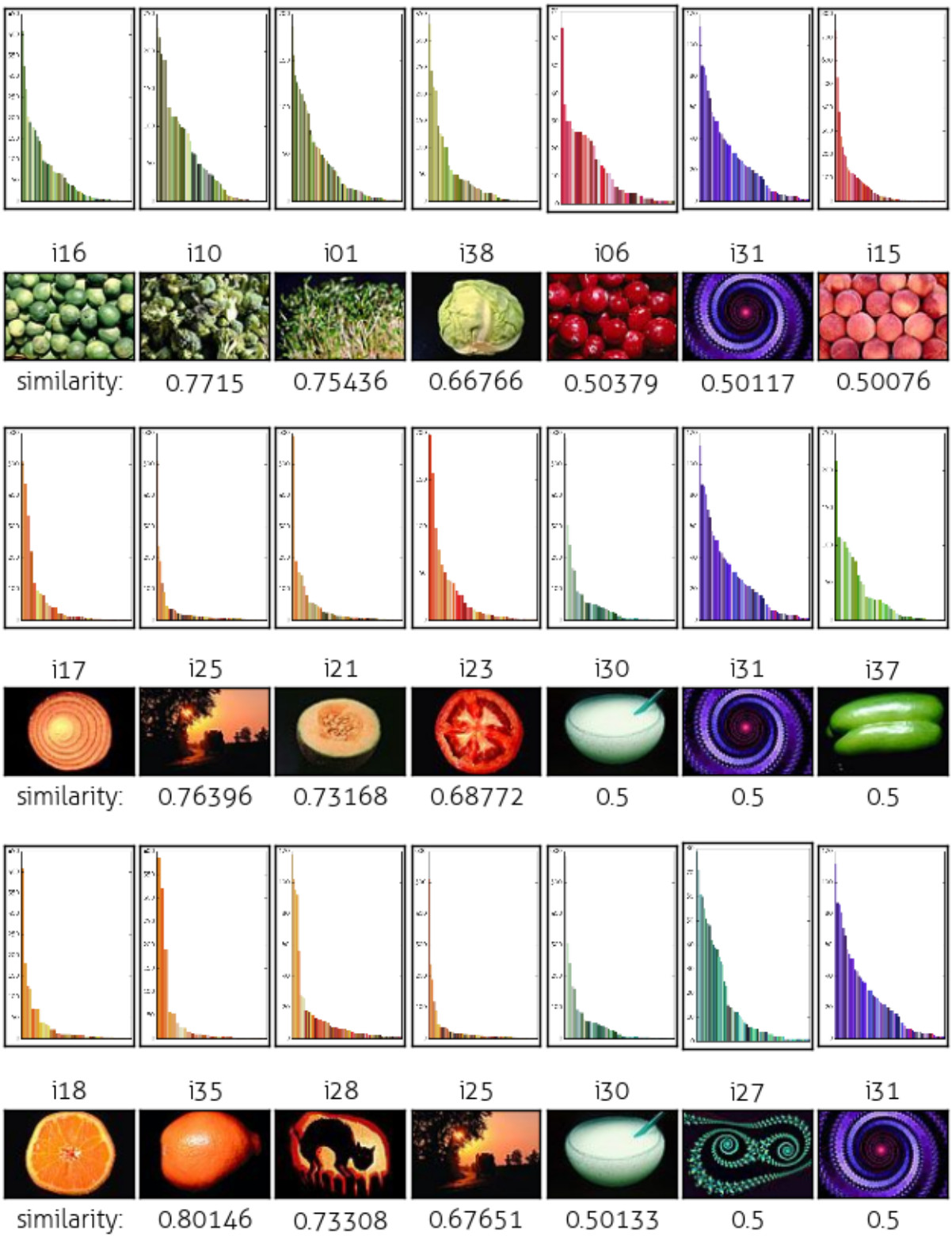


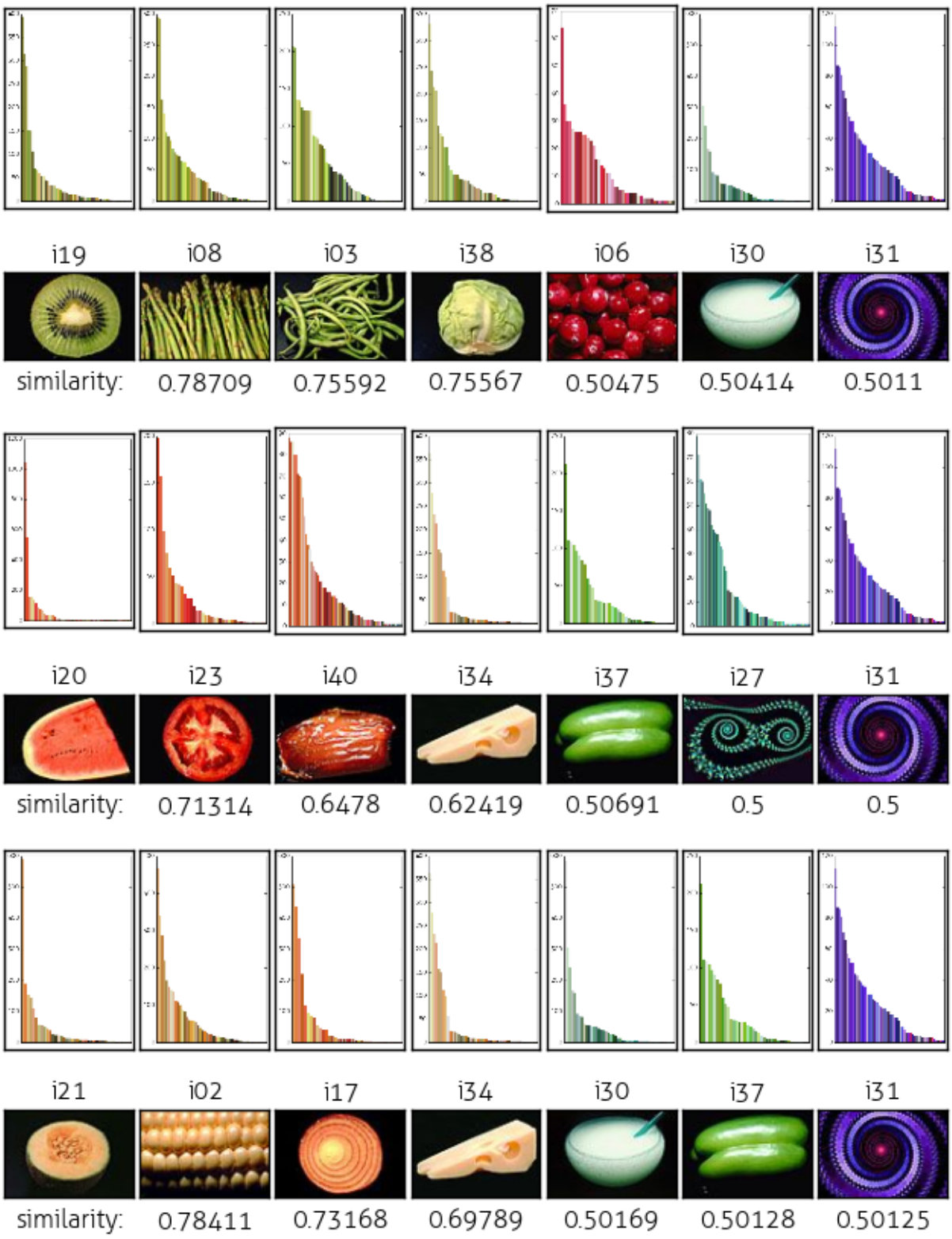

i04 i03 i36 i24 i06 i31 i15
similarity: 0.85601 0.74297 0.74101 0.50168 0.50155 0.5003
i05 i03 i01 i24 i26 i30 i31
similarity: 0.66707 0.66349 0.65765 0.52189 0.51823 0.50485
i06 i11 i05 i40 i04 i03 i27
similarity: 0.64935 0.63072 0.62789 0.50168 0.50142 0.5

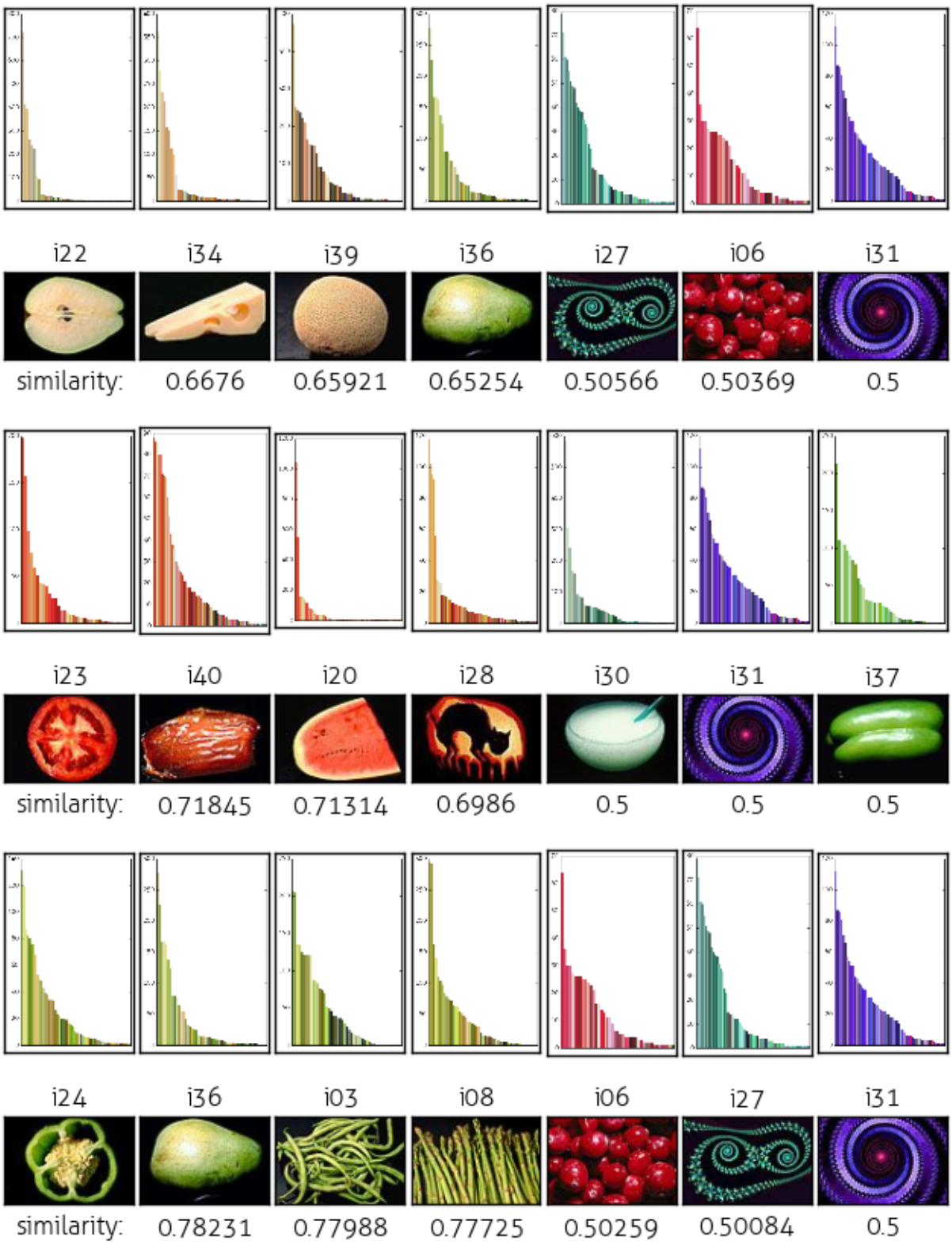

i07 i40 i09 i11 i31 i27 i30
similarity: 0.76591 0.74977 0.7123 0.50291 0.50266 0.50182
i08 i19 i24 i03 i06 i30 i31
similarity: 0.78709 0.77725 0.77411 0.5025 0.50086 0.50027
i09 i07 i11 i14 i32 i27 i31
similarity: 0.74977 0.74072 0.73429 0.51241 0.5057 0.50238

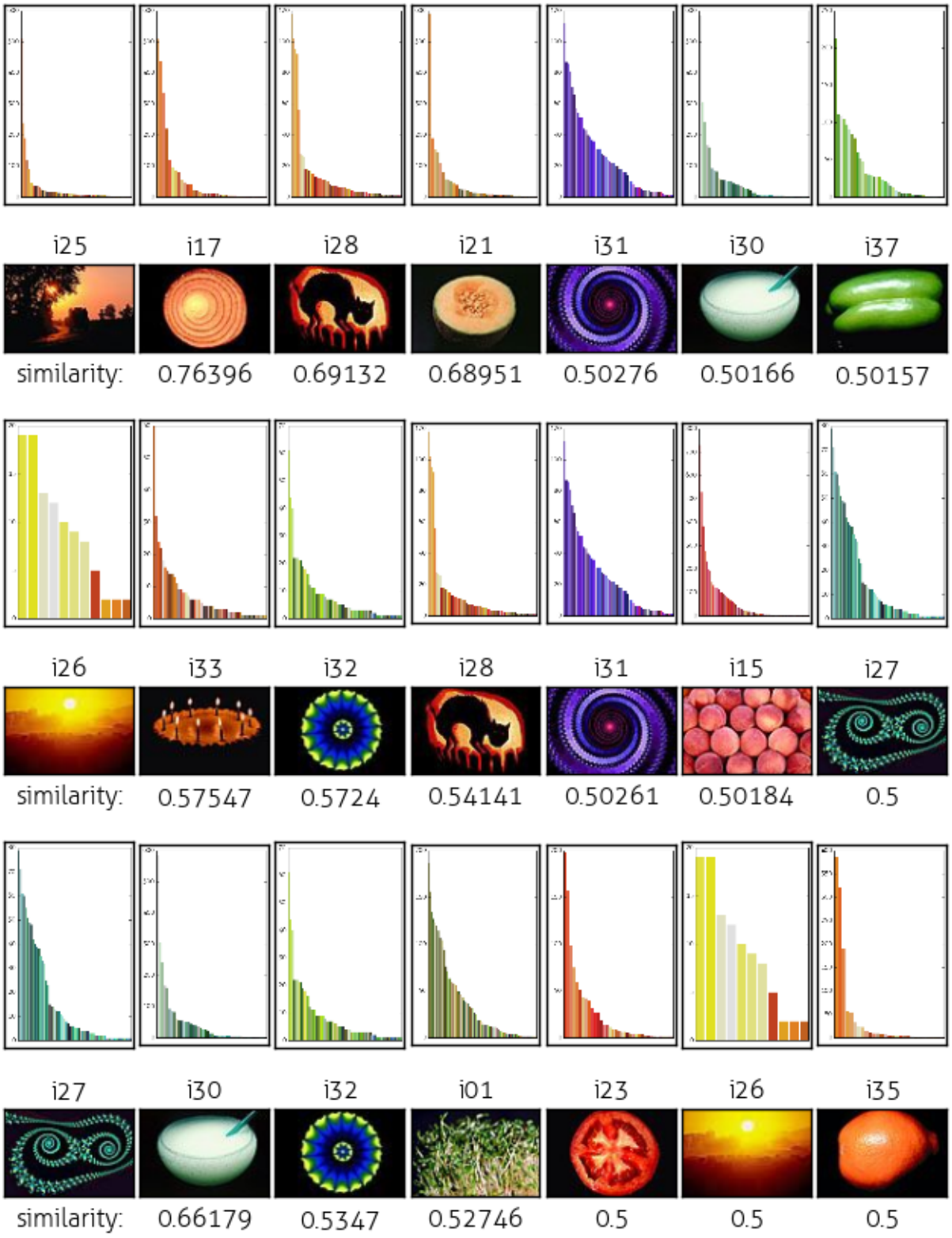

i10 i01 i16 i03 i15 i31 i17
similarity: 0.84572 0.7715 0.69472 0.50166 0.50136 0.50115
i11 i09 i07 i05 i37 i26 i31
similarity: 0.74072 0.7123 0.65142 0.51175 0.5093 0.5059
i12 i09 i02 i39 i32 i31 i27
similarity: 0.67196 0.66583 0.66009 0.51007 0.50213 0.50211

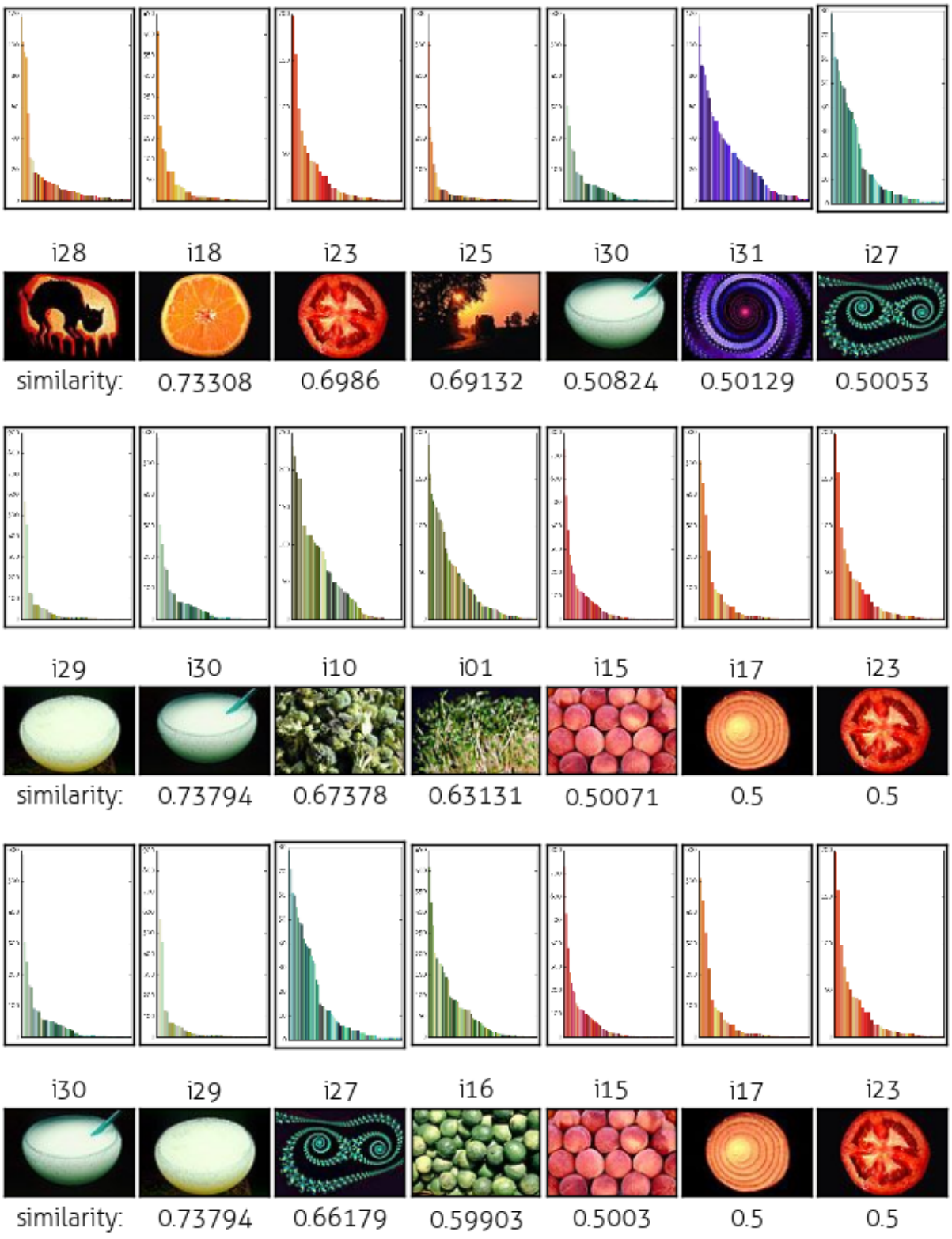

i13 i10 i14 i01 i31 i23 i17
similarity: 0.68352 0.66104 0.63021 0.50236 0.50146 0.5003
i14 i09 i39 i13 i32 i26 i31
similarity: 0.73429 0.72073 0.66104 0.50632 0.50583 0.5016
i15 i07 i40 i06 i04 i32 i27
similarity: 0.70467 0.61896 0.59795 0.5003 0.50021 0.50019

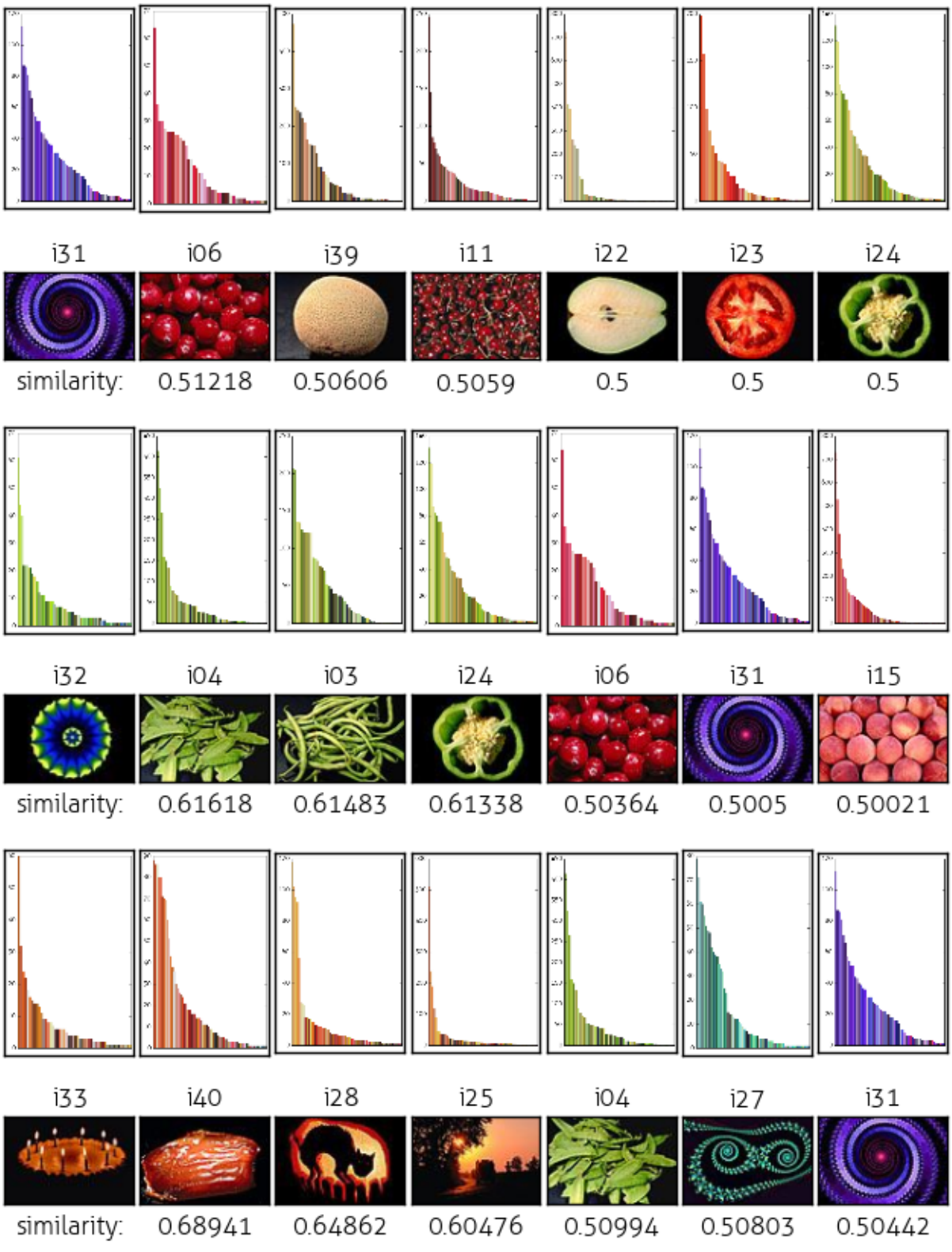

i16 i10 i01 i38 i06 i31 i15
similarity: 0.7715 0.75436 0.66766 0.50379 0.50117 0.50076
i17 i25 i21 i23 i30 i31 i37
similarity: 0.76396 0.73168 0.68772 0.5 0.5 0.5
i18 i35 i28 i25 i30 i27 i31
similarity: 0.80146 0.73308 0.67651 0.50133 0.5 0.5

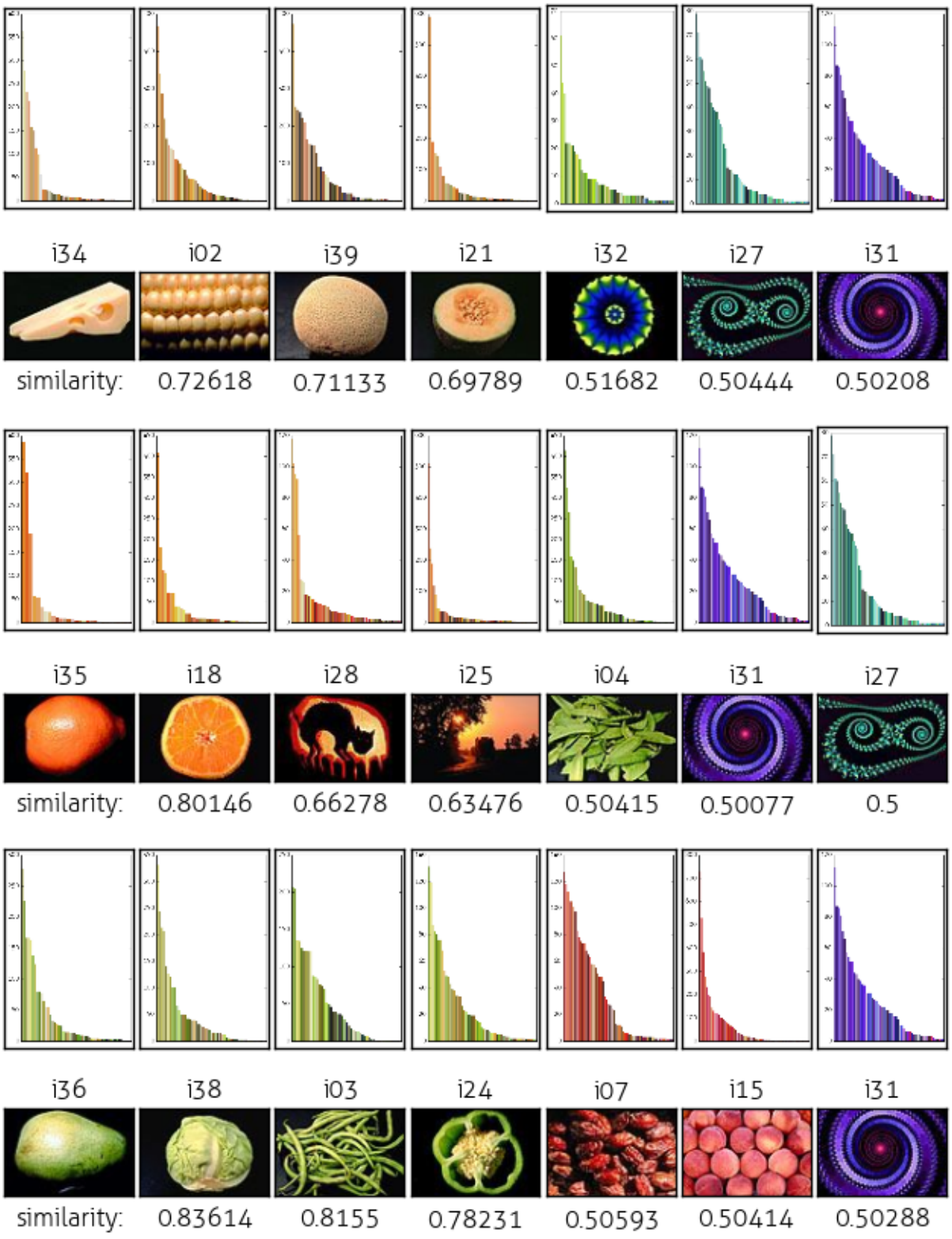

i19 i08 i03 i38 i06 i30 i31
similarity: 0.78709 0.75592 0.75567 0.50475 0.50414 0.5011
i20 i23 i40 i34 i37 i27 i31
similarity: 0.71314 0.6478 0.62419 0.50691 0.5 0.5
i21 i02 i17 i34 i30 i37 i31
similarity: 0.78411 0.73168 0.69789 0.50169 0.50128 0.50125

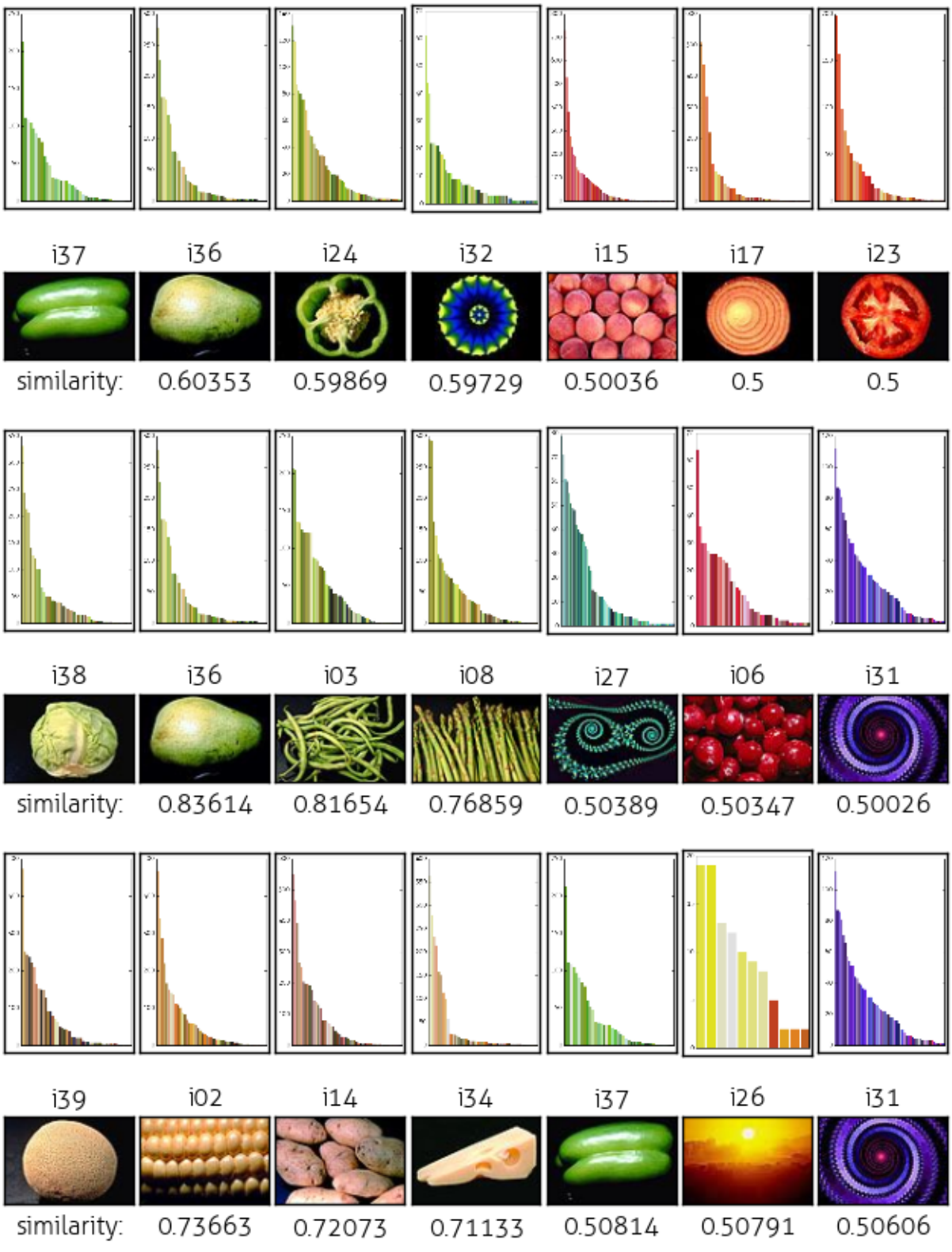

i22 i34 i39 i36 i27 i06 i31
similarity: 0.6676 0.65921 0.65254 0.50566 0.50369 0.5
i23 i40 i20 i28 i30 i31 i37
similarity: 0.71845 0.71314 0.6986 0.5 0.5 0.5
i24 i36 i03 i08 i06 i27 i31
similarity: 0.78231 0.77988 0.77725 0.50259 0.50084 0.5

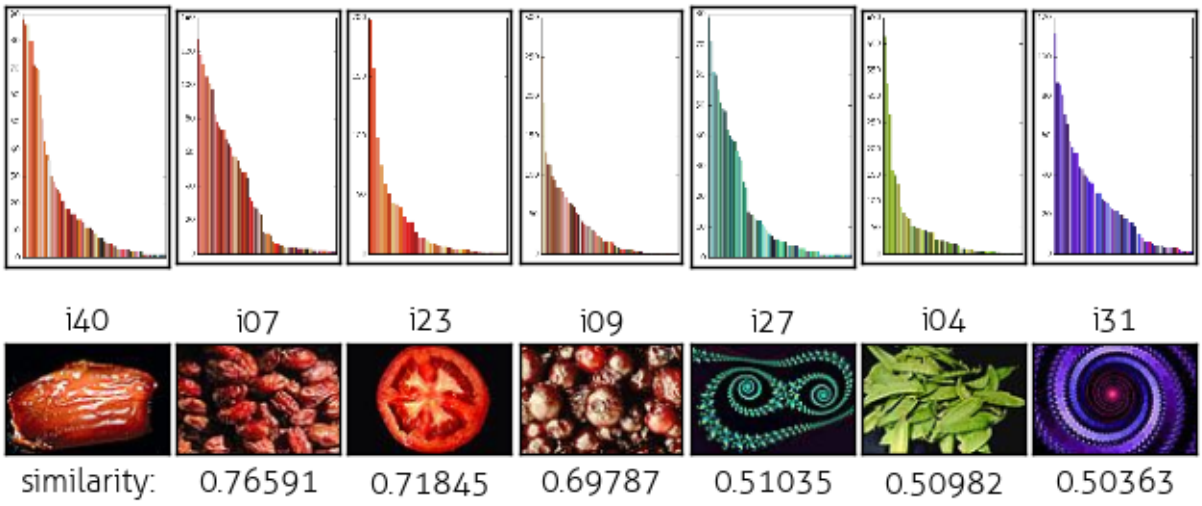

i25 i17 i28 i21 i31 i30 i37
similarity: 0.76396 0.69132 0.68951 0.50276 0.50166 0.50157
i26 i33 i32 i28 i31 i15 i27
similarity: 0.57547 0.5724 0.54141 0.50261 0.50184 0.5
i27 i30 i32 i01 i23 i26 i35
similarity: 0.66179 0.5347 0.52746 0.5 0.5 0.5


i28 i18 i23 i25 i30 i31 i27
similarity: 0.73308 0.6986 0.69132 0.50824 0.50129 0.50053
i29 i30 i10 i01 i15 i17 i23
similarity: 0.73794 0.67378 0.63131 0.50071 0.5 0.5
i30 i29 i27 i16 i15 i17 i23
similarity: 0.73794 0.66179 0.59903 0.5003 0.5 0.5

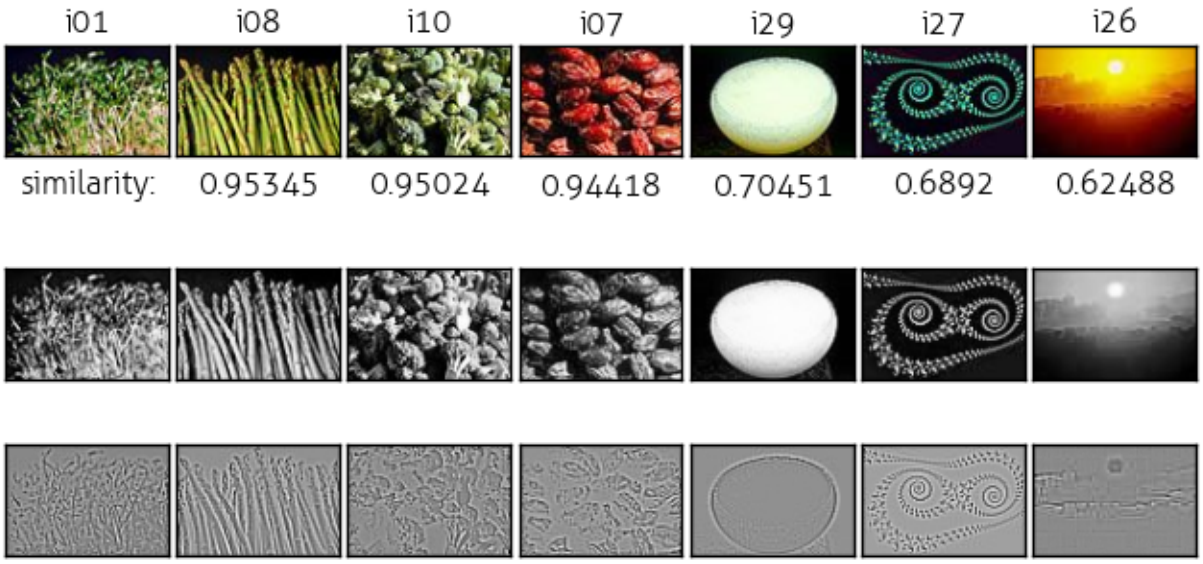

i31 i06 i39 i11 i22 i23 i24
similarity: 0.51218 0.50606 0.5059 0.5 0.5 0.5
i32 i04 i03 i24 i06 i31 i15
similarity: 0.61618 0.61483 0.61338 0.50364 0.5005 0.50021
i33 i40 i28 i25 i04 i27 i31
similarity: 0.68941 0.64862 0.60476 0.50994 0.50803 0.50442

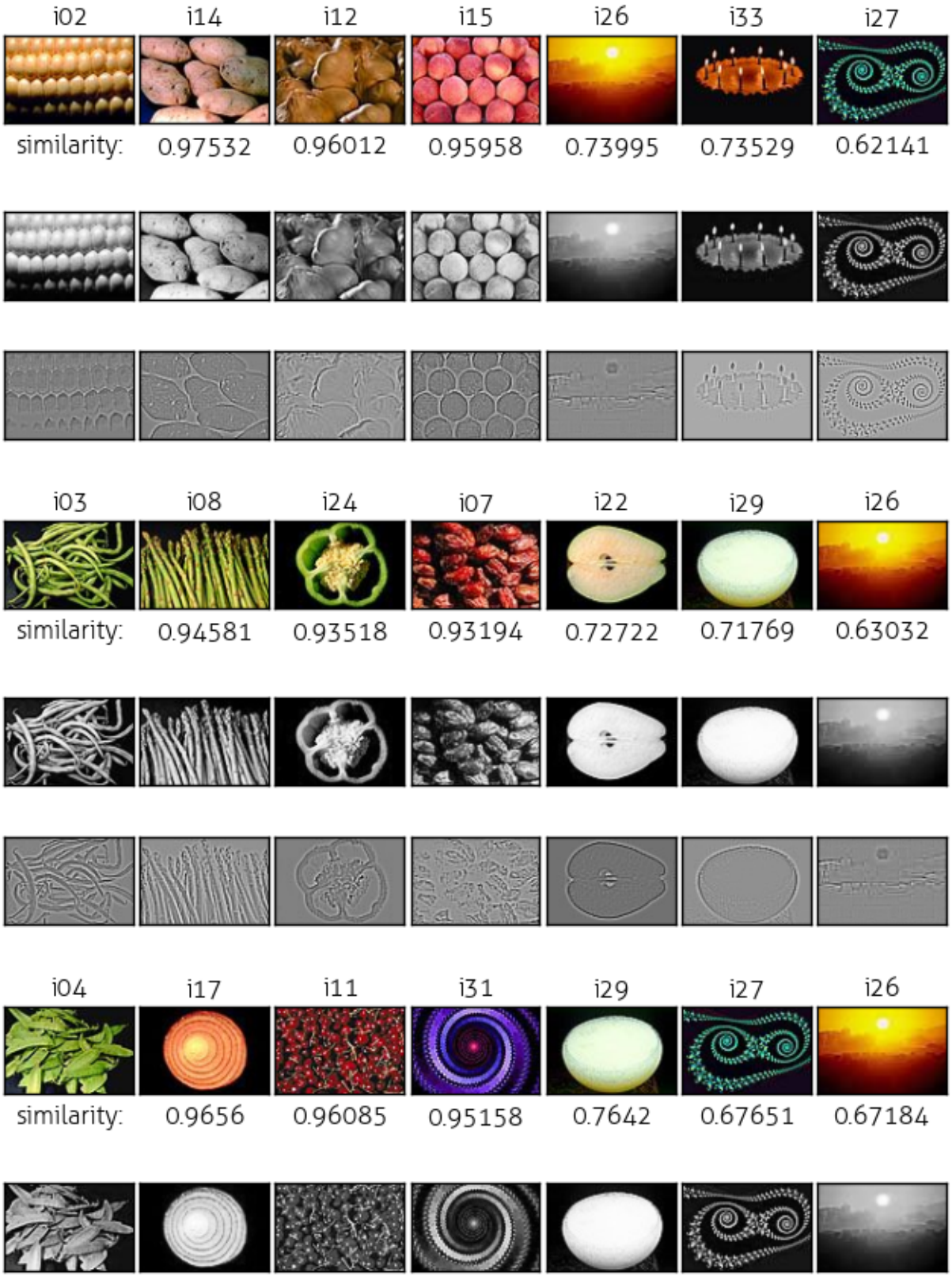

i34 i02 i39 i21 i32 i27 i31
similarity: 0.72618 0.71133 0.69789 0.51682 0.50444 0.50208
i35 i18 i28 i25 i04 i31 i27
similarity: 0.80146 0.66278 0.63476 0.50415 0.50077 0.5
i36 i38 i03 i24 i07 i15 i31
similarity: 0.83614 0.8155 0.78231 0.50593 0.50414 0.50288

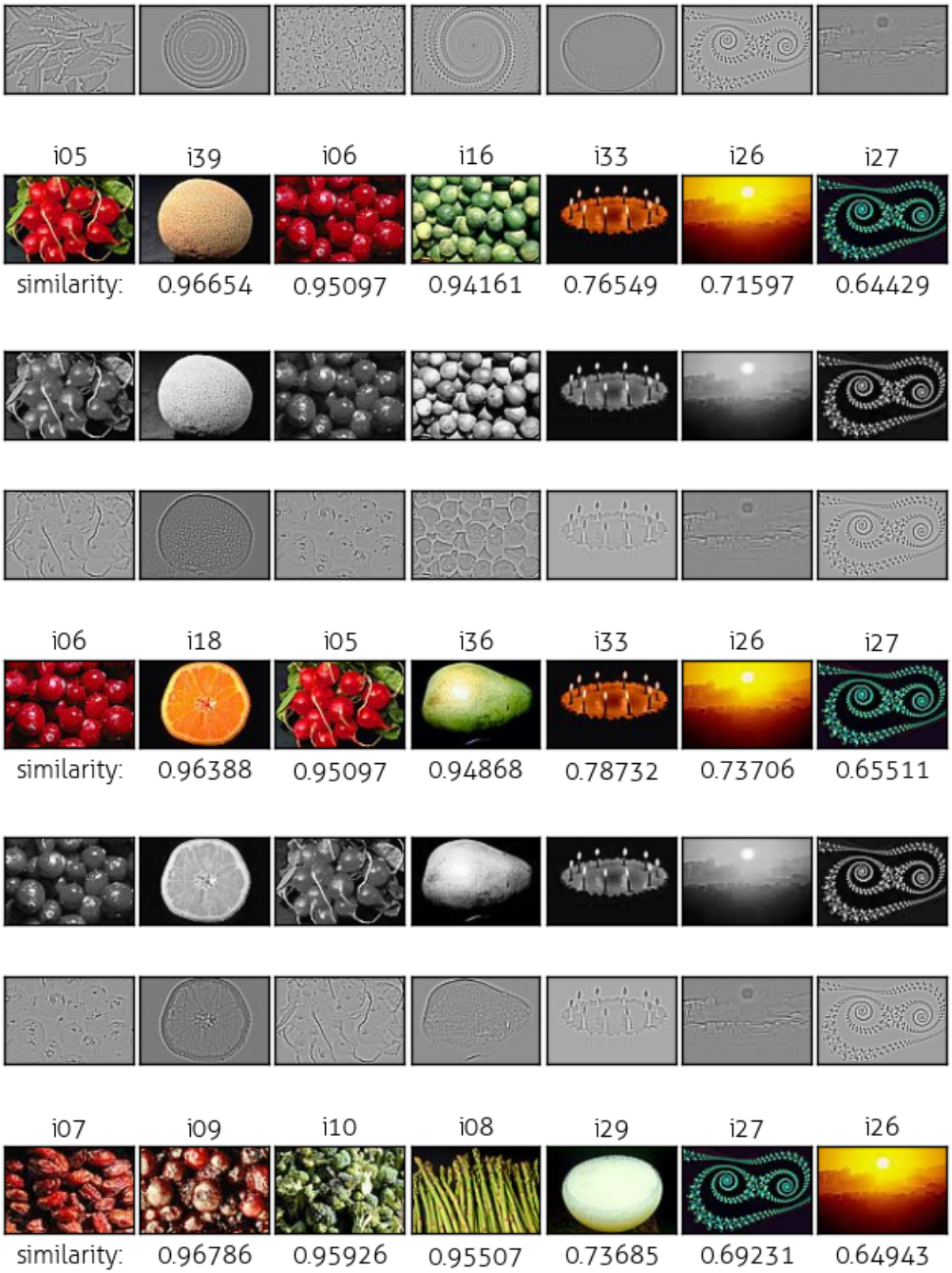

i37 i36 i24 i32 i15 i17 i23
similarity: 0.60353 0.59869 0.59729 0.50036 0.5 0.5
i38 i36 i03 i08 i27 i06 i31
similarity: 0.83614 0.81654 0.76859 0.50389 0.50347 0.50026
i39 i02 i14 i34 i37 i26 i31
similarity: 0.73663 0.72073 0.71133 0.50814 0.50791 0.50606

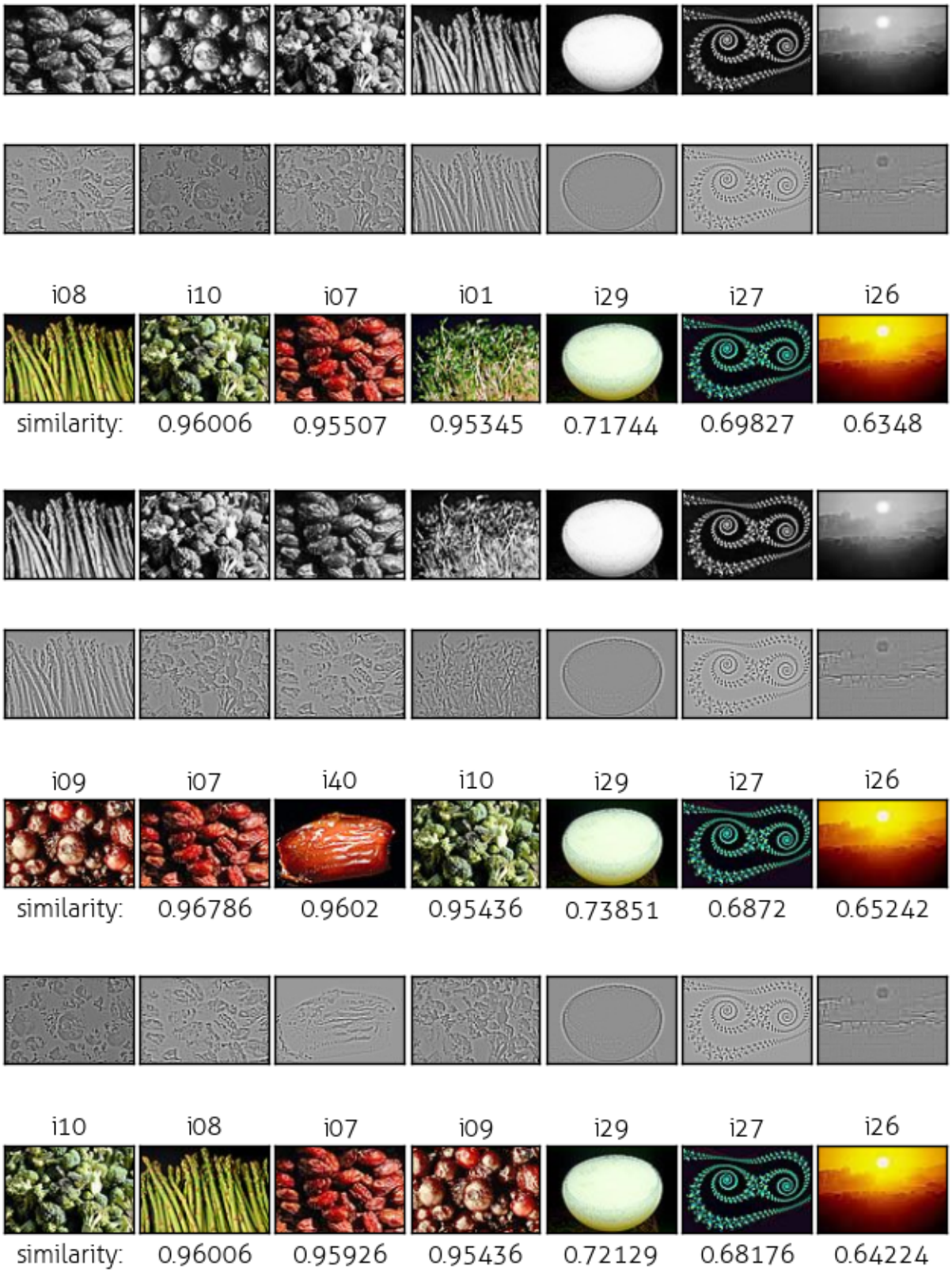

i40
i07
i23
i09
i27
i04
i31
similarity:
0.76591
0.71845
0.69787
0.51035
0.50982
0.50363

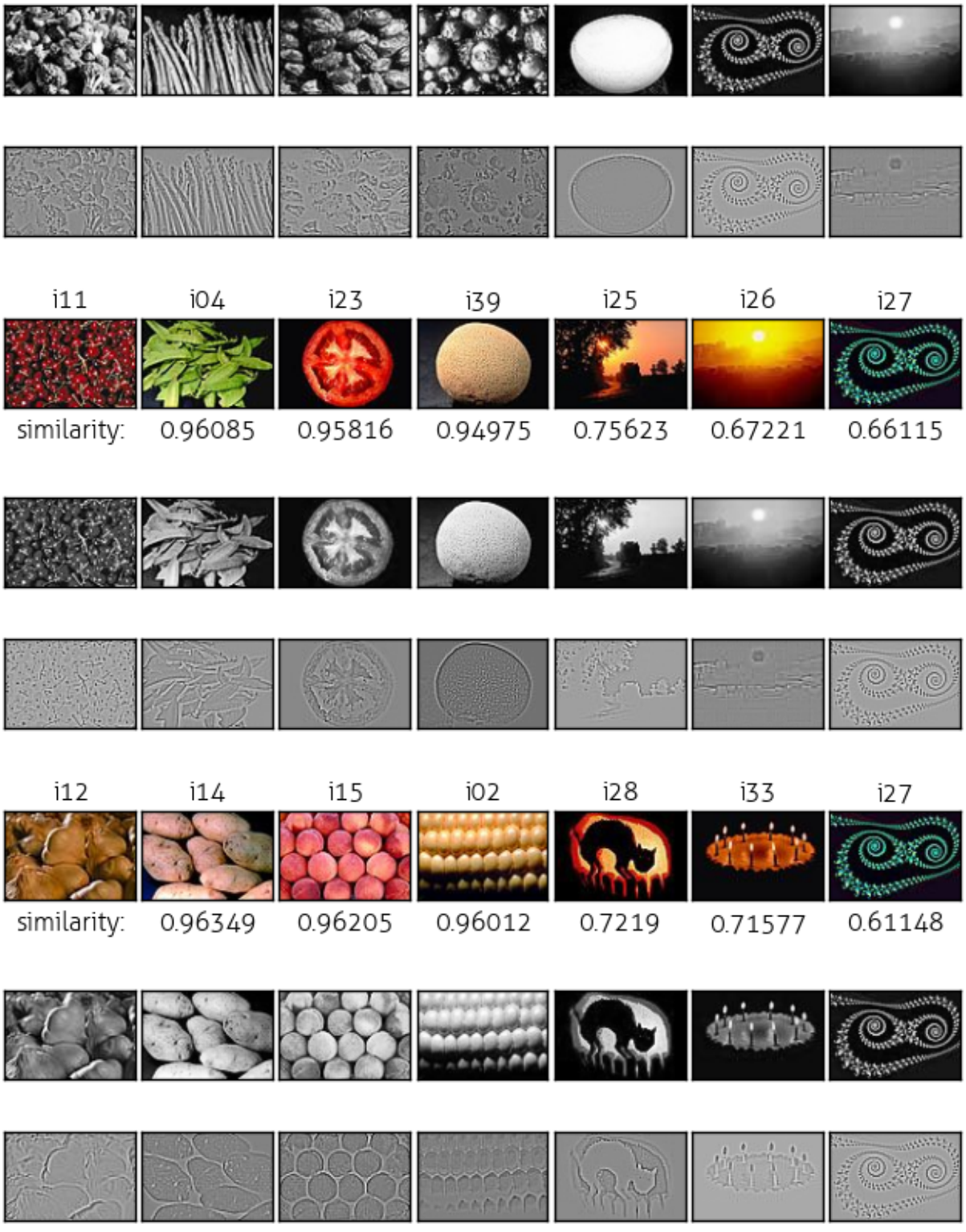
Appendix B: Septuples + Laplacian Images

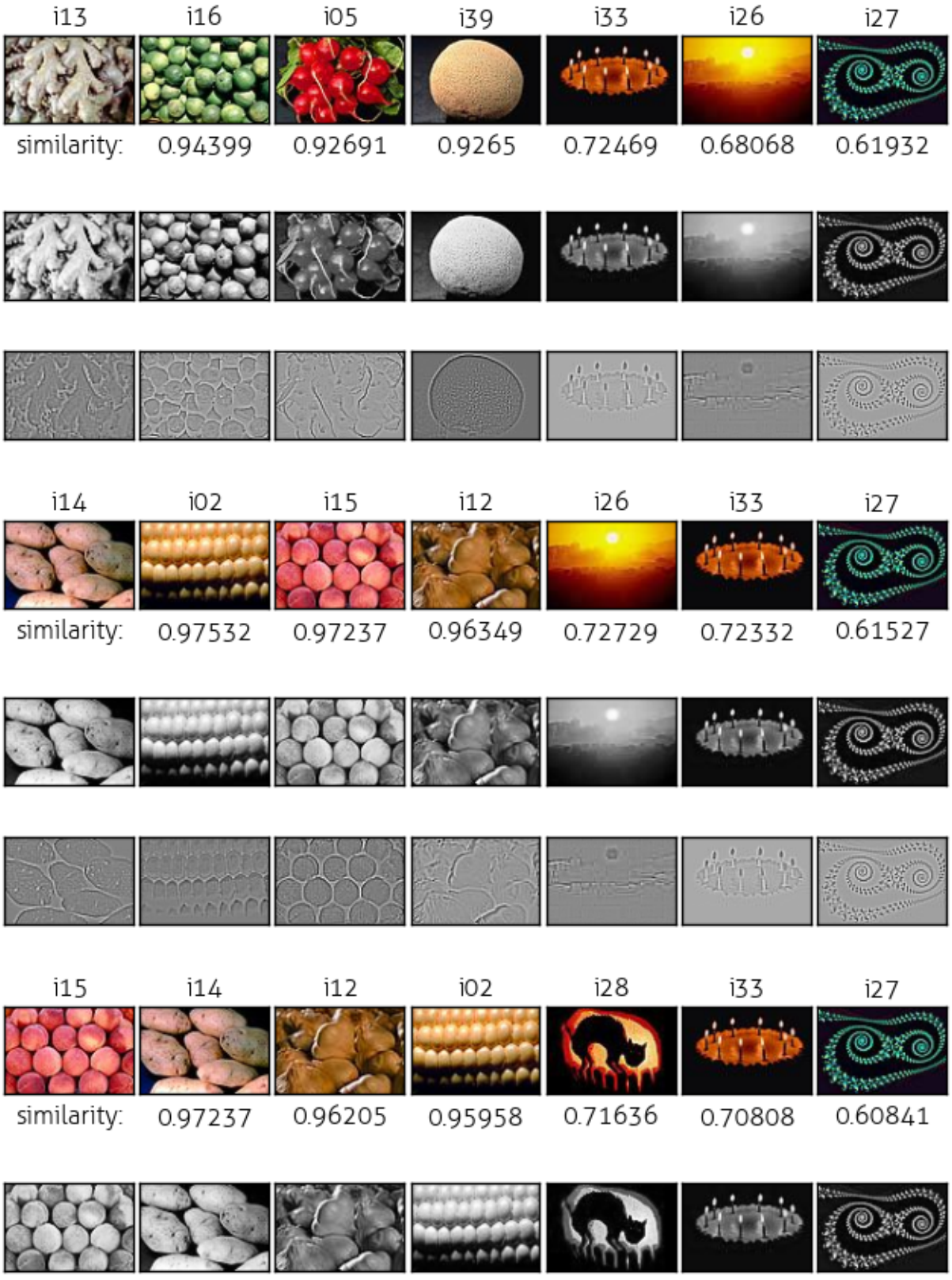


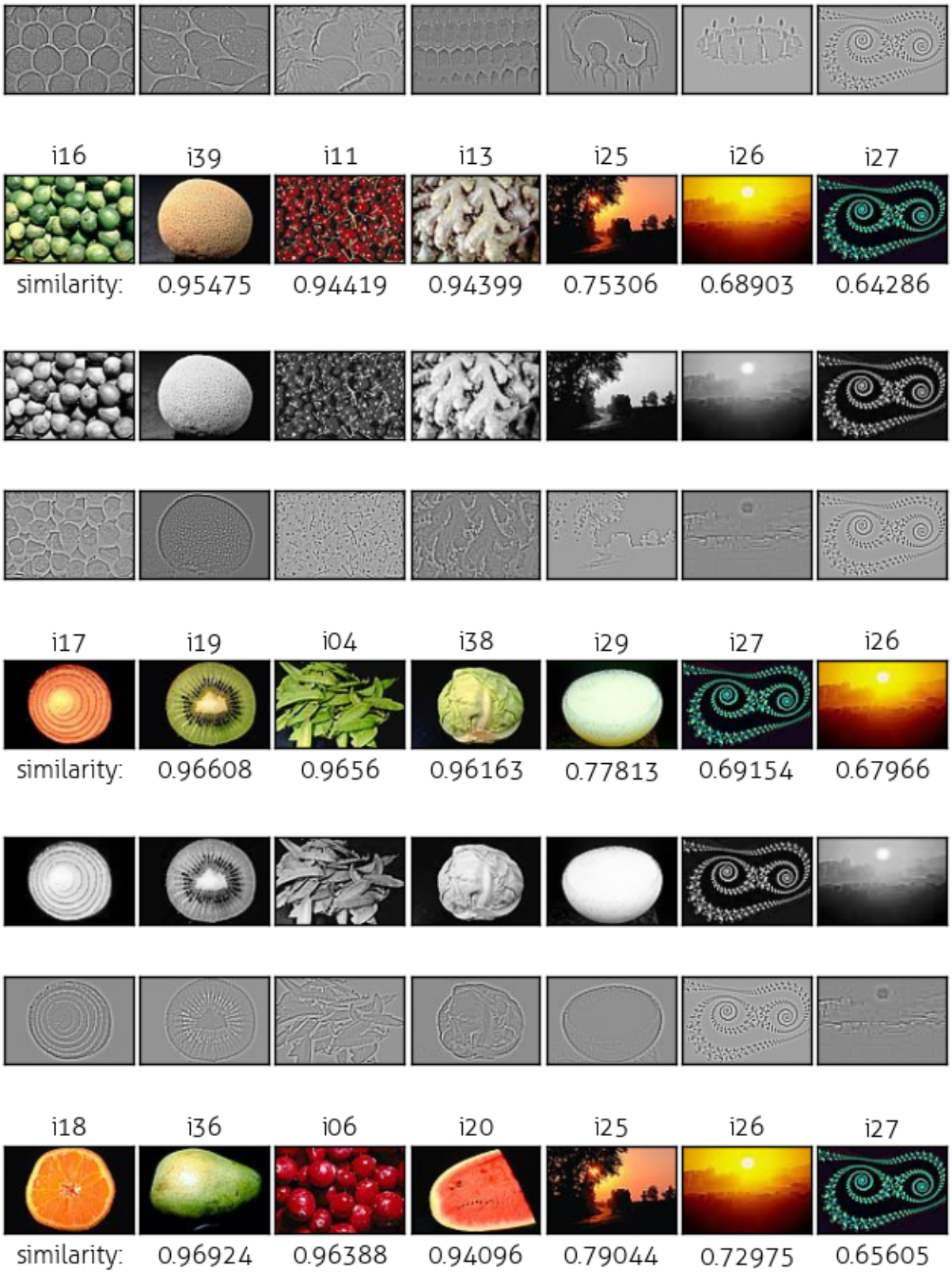

i02 i14 i12 i15 i26 i33 i27
similarity: 0.97532 0.96012 0.95958 0.73995 0.73529 0.62141
i03 i08 i24 i07 i22 i29 i26
similarity: 0.94581 0.93518 0.93194 0.72722 0.71769 0.63032
i04 i17 i11 i31 i29 i27 i26
similarity: 0.9656 0.96085 0.95158 0.7642 0.67651 0.67184

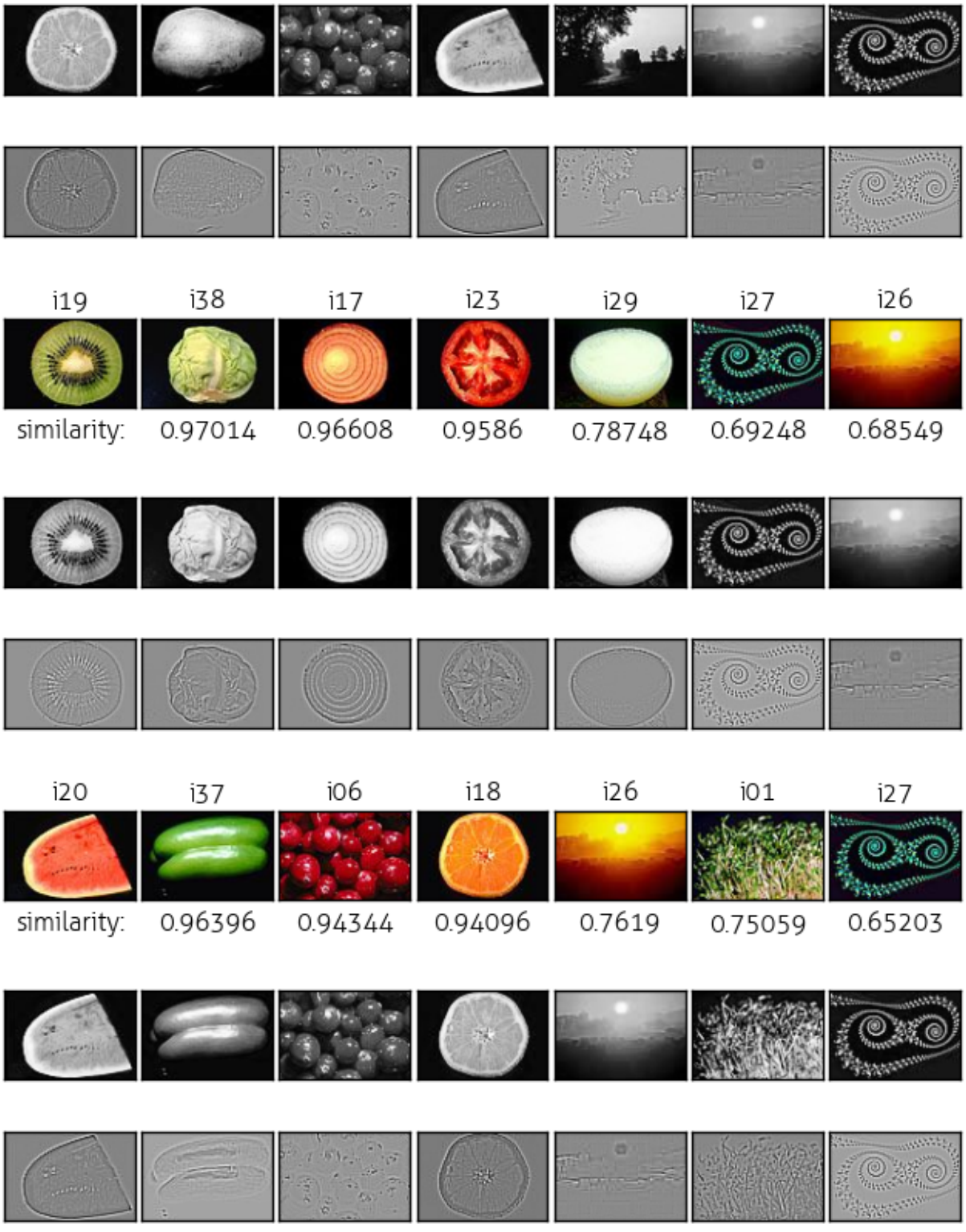

i05 i39 i06 i16 i33 i26 i27
similarity: 0.96654 0.95097 0.94161 0.76549 0.71597 0.64429
i06 i18 i05 i36 i33 i26 i27
similarity: 0.96388 0.95097 0.94868 0.78732 0.73706 0.65511
i07 i09 i10 i08 i29 i27 i26
similarity: 0.96786 0.95926 0.95507 0.73685 0.69231 0.64943

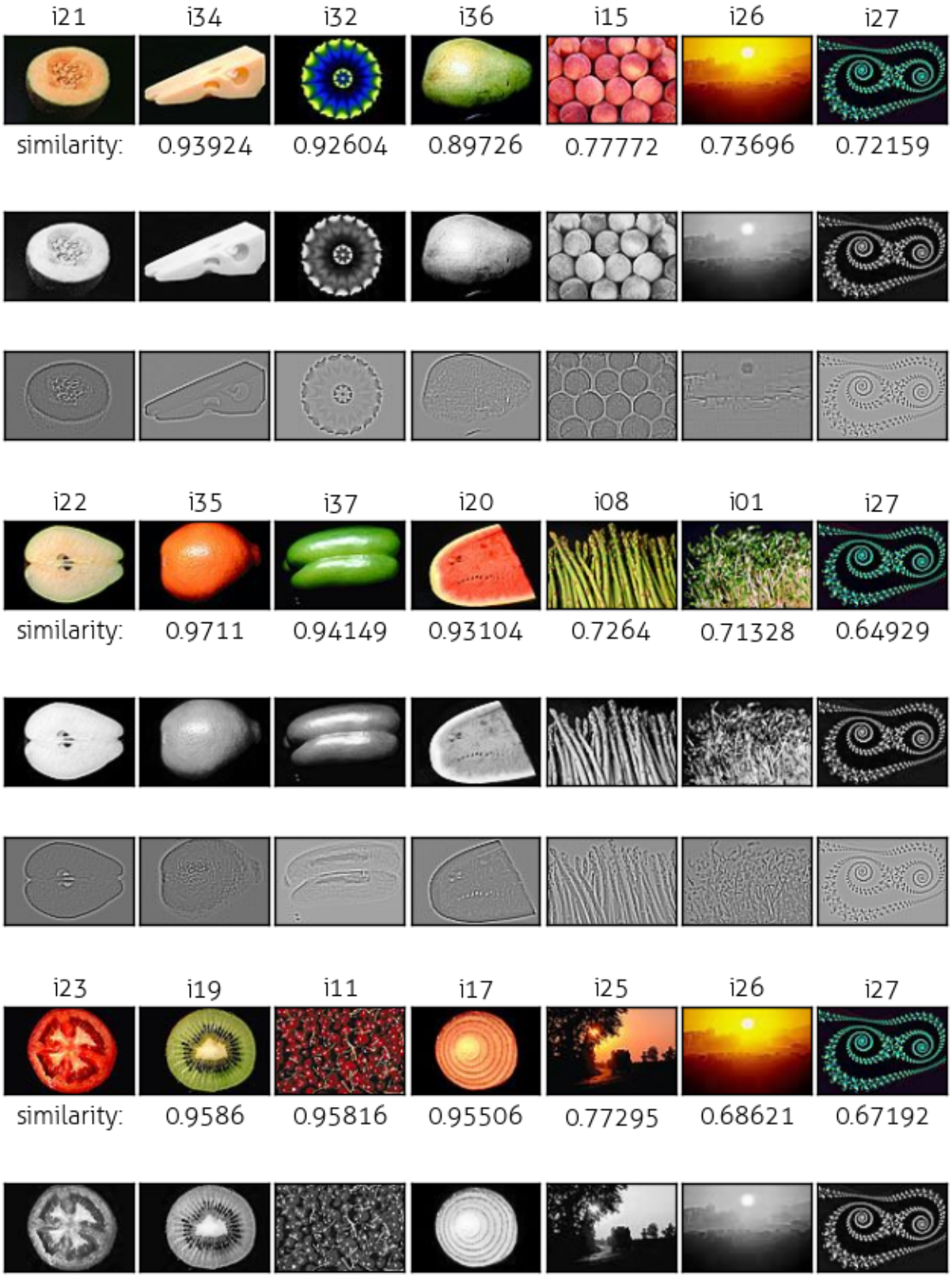

i08 i10 i07 i01 i29 i27 i26
similarity: 0.96006 0.95507 0.95345 0.71744 0.69827 0.6348
i09 i07 i40 i10 i29 i27 i26
similarity: 0.96786 0.9602 0.95436 0.73851 0.6872 0.65242
i10 i08 i07 i09 i29 i27 i26
similarity: 0.96006 0.95926 0.95436 0.72129 0.68176 0.64224

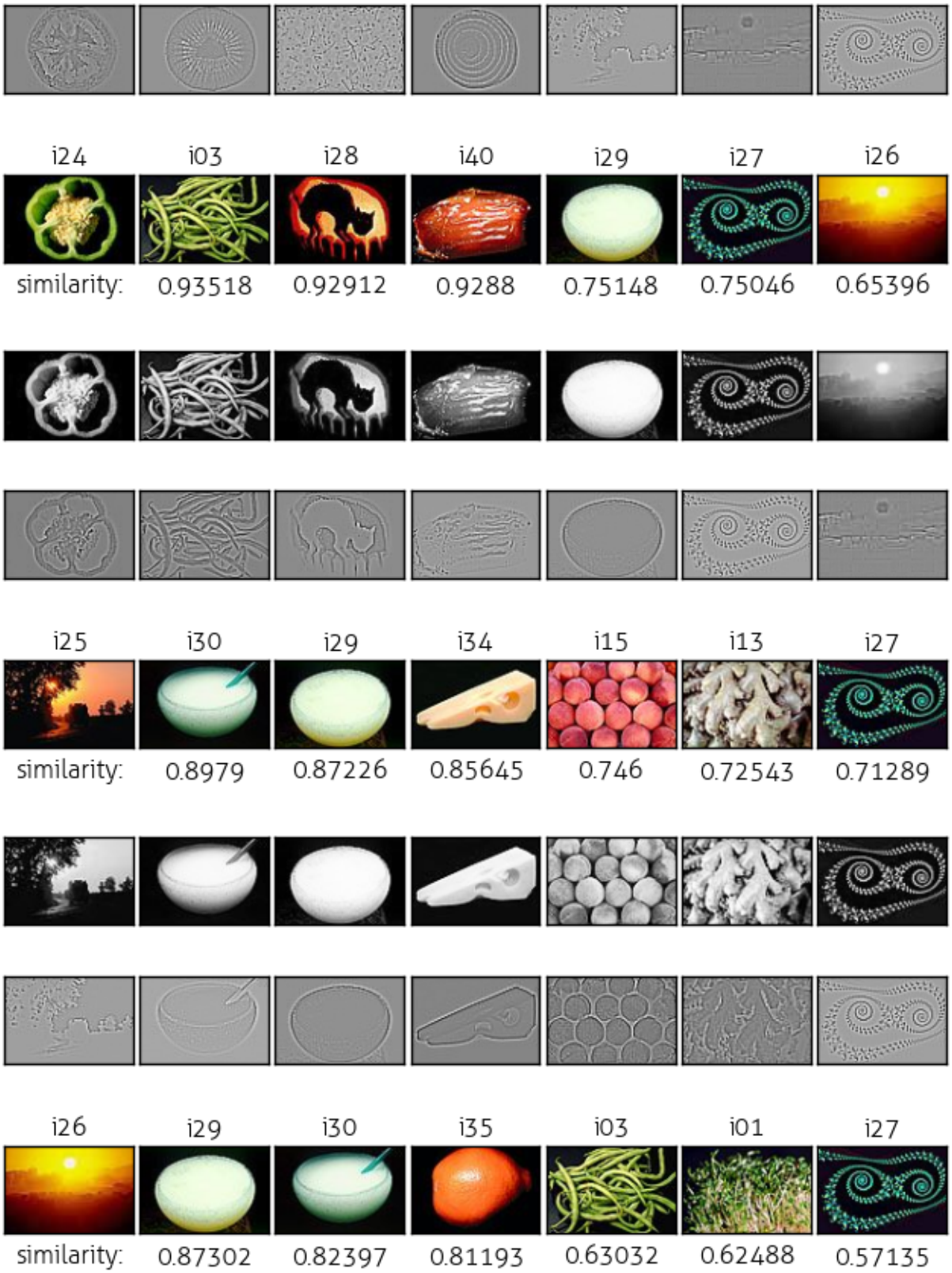

i11 i04 i23 i39 i25 i26 i27
similarity: 0.96085 0.95816 0.94975 0.75623 0.67221 0.66115
i12 i14 i15 i02 i28 i33 i27
similarity: 0.96349 0.96205 0.96012 0.7219 0.71577 0.61148

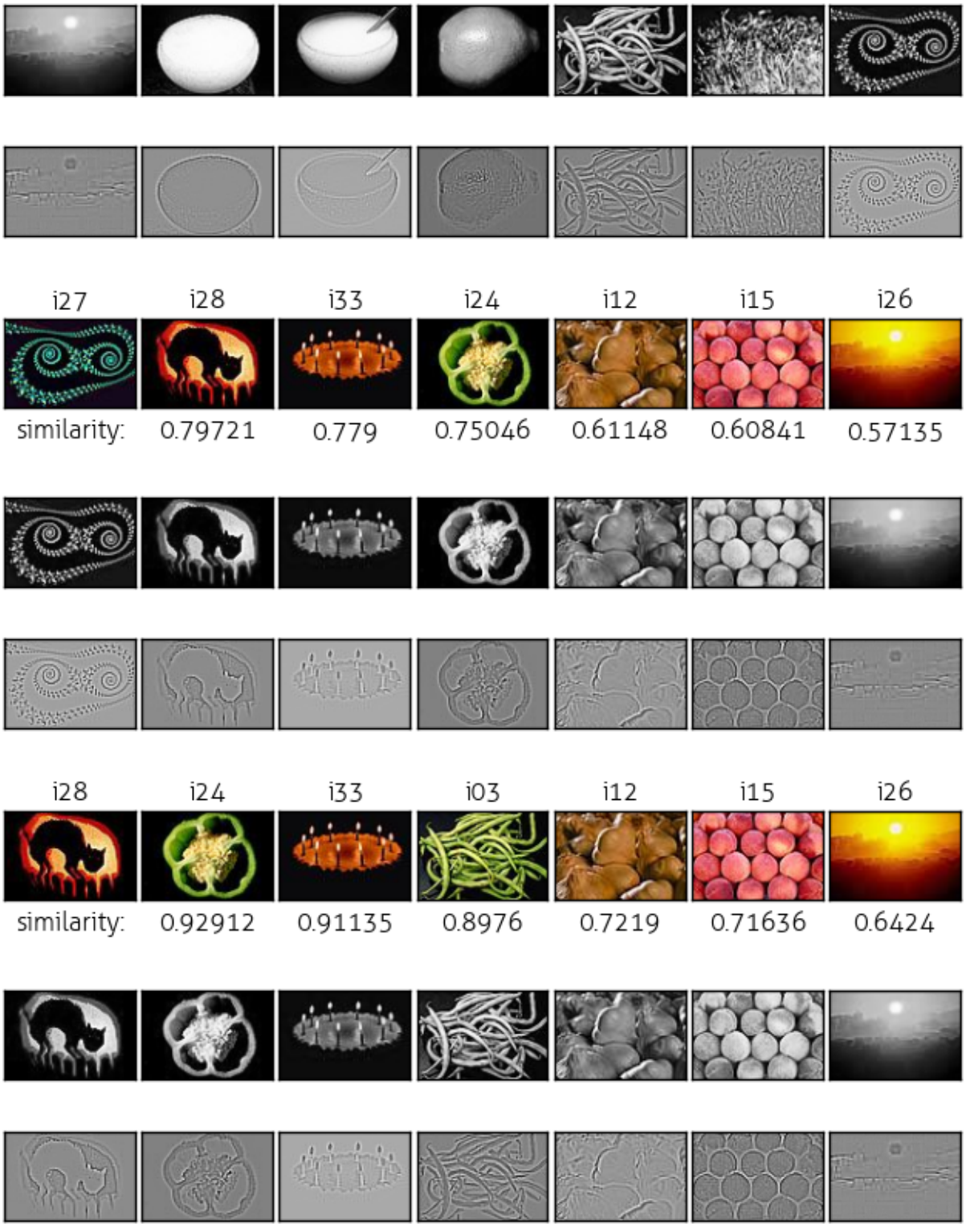

i13 i16 i05 i39 i33 i26 i27
similarity: 0.94399 0.92691 0.9265 0.72469 0.68068 0.61932
i14 i02 i15 i12 i26 i33 i27
similarity: 0.97532 0.97237 0.96349 0.72729 0.72332 0.61527
i15 i14 i12 i02 i28 i33 i27
similarity: 0.97237 0.96205 0.95958 0.71636 0.70808 0.60841

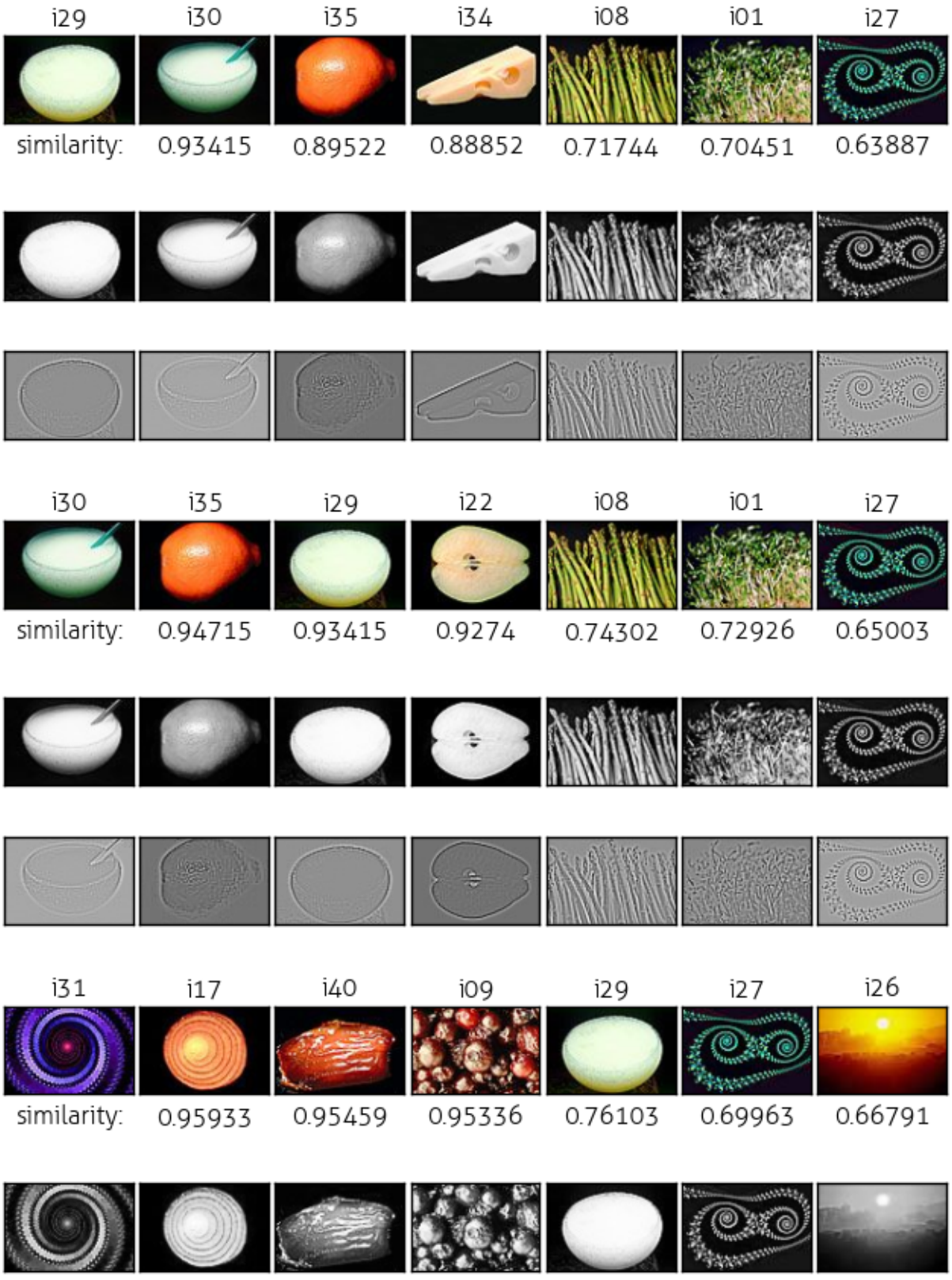

i16 i39 i11 i13 i25 i26 i27
similarity: 0.95475 0.94419 0.94399 0.75306 0.68903 0.64286
i17 i19 i04 i38 i29 i27 i26
similarity: 0.96608 0.9656 0.96163 0.77813 0.69154 0.67966
i18 i36 i06 i20 i25 i26 i27
similarity: 0.96924 0.96388 0.94096 0.79044 0.72975 0.65605

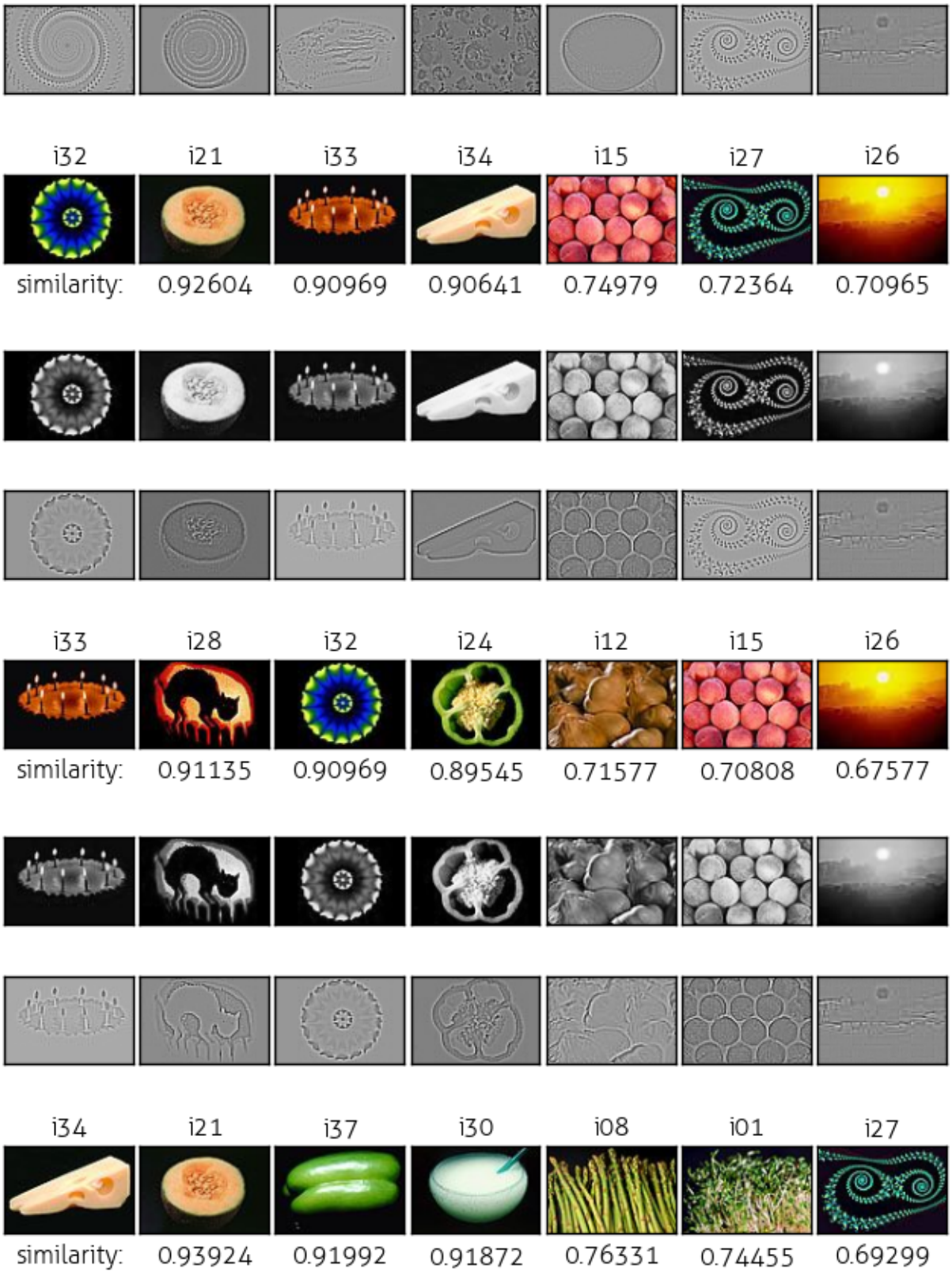

i19 i38 i17 i23 i29 i27 i26
similarity: 0.97014 0.96608 0.9586 0.78748 0.69248 0.68549
i20 i37 i06 i18 i26 i01 i27
similarity: 0.96396 0.94344 0.94096 0.7619 0.75059 0.65203

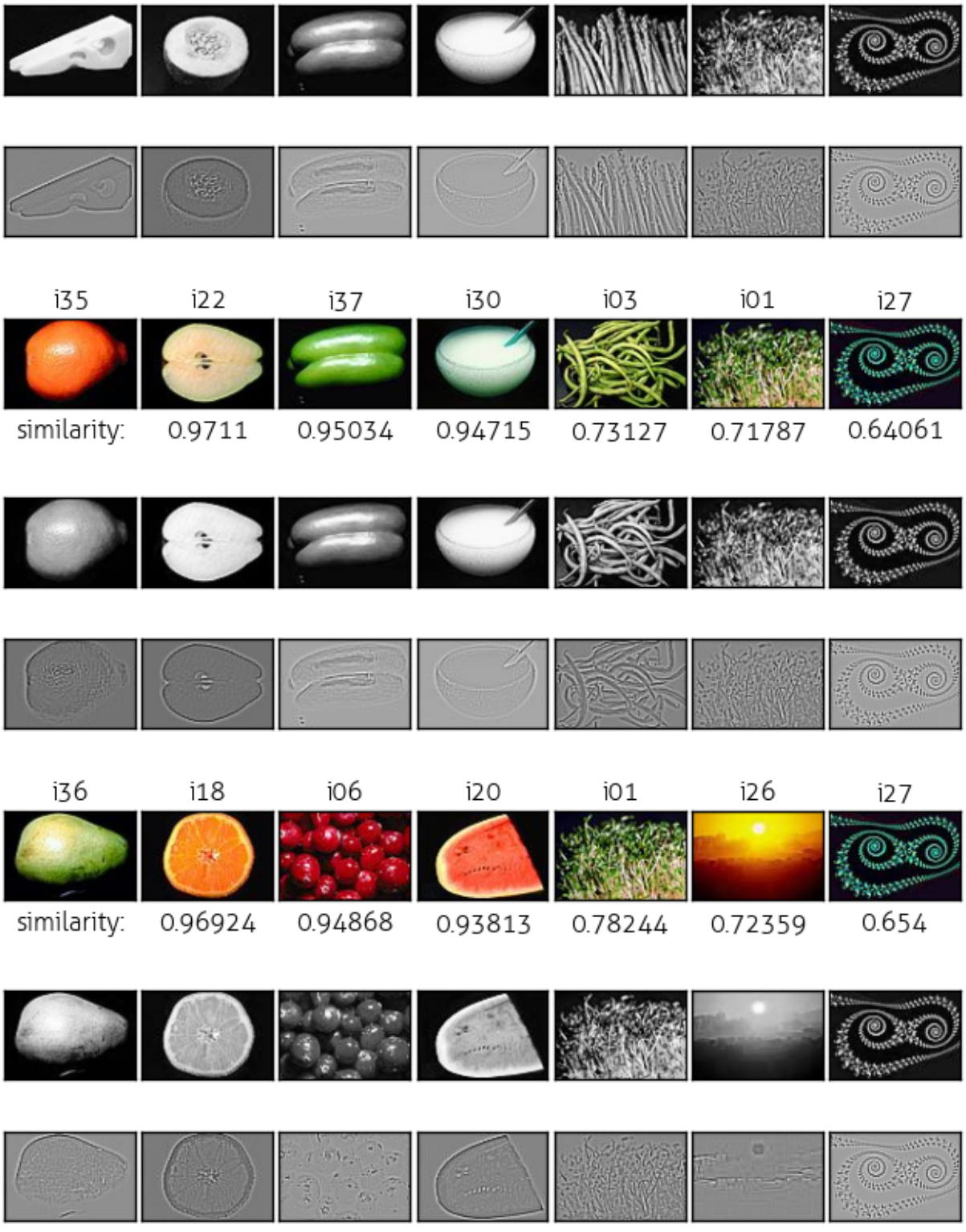

i21 i34 i32 i36 i15 i26 i27
similarity: 0.93924 0.92604 0.89726 0.77772 0.73696 0.72159
i22 i35 i37 i20 i08 i01 i27
similarity: 0.9711 0.94149 0.93104 0.7264 0.71328 0.64929
i23 i19 i11 i17 i25 i26 i27
similarity: 0.9586 0.95816 0.95506 0.77295 0.68621 0.67192

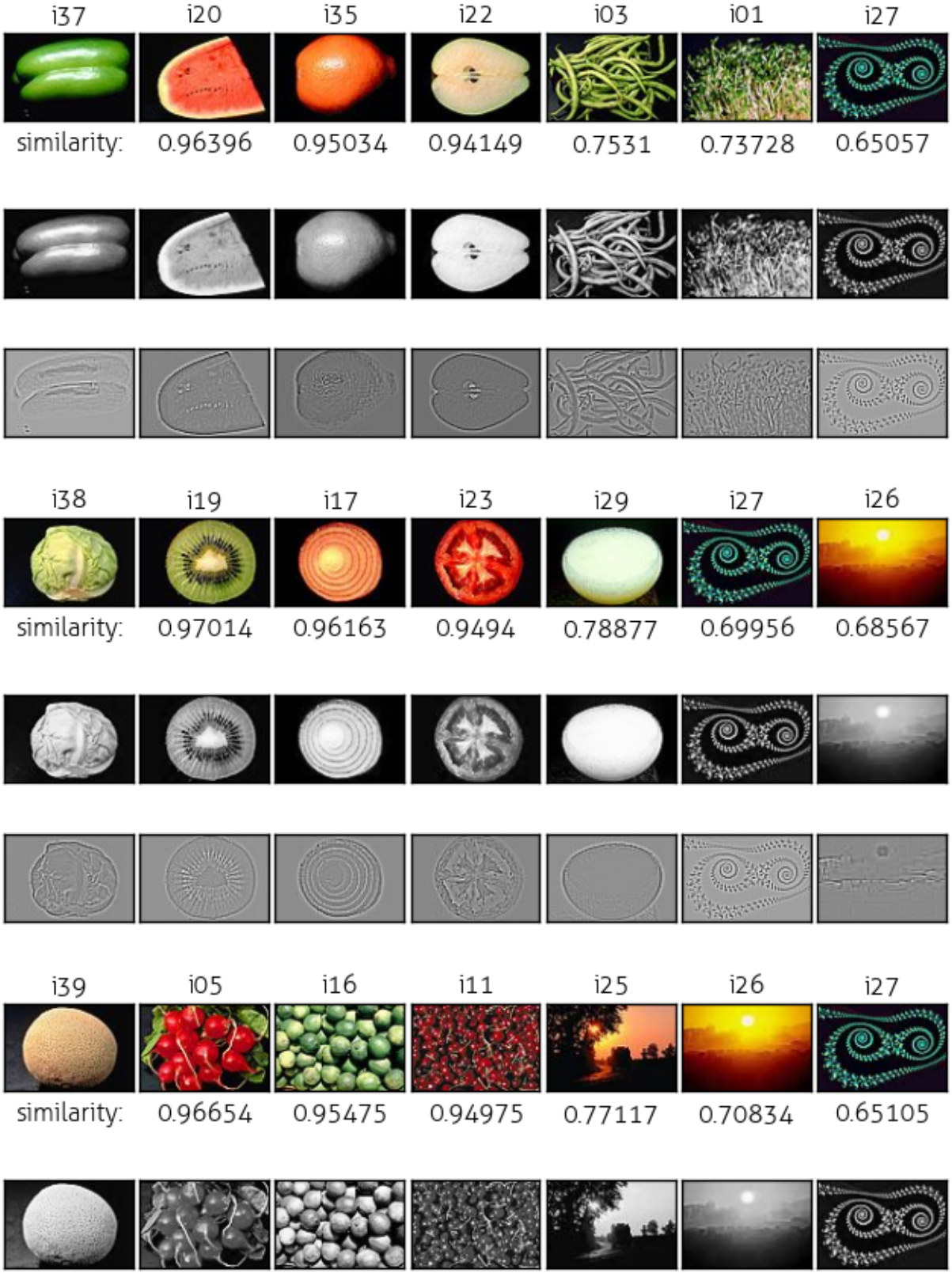

i24 i03 i28 i40 i29 i27 i26
similarity: 0.93518 0.92912 0.9288 0.75148 0.75046 0.65396
i25 i30 i29 i34 i15 i13 i27
similarity: 0.8979 0.87226 0.85645 0.746 0.72543 0.71289
i26 i29 i30 i35 i03 i01 i27
similarity: 0.87302 0.82397 0.81193 0.63032 0.62488 0.57135

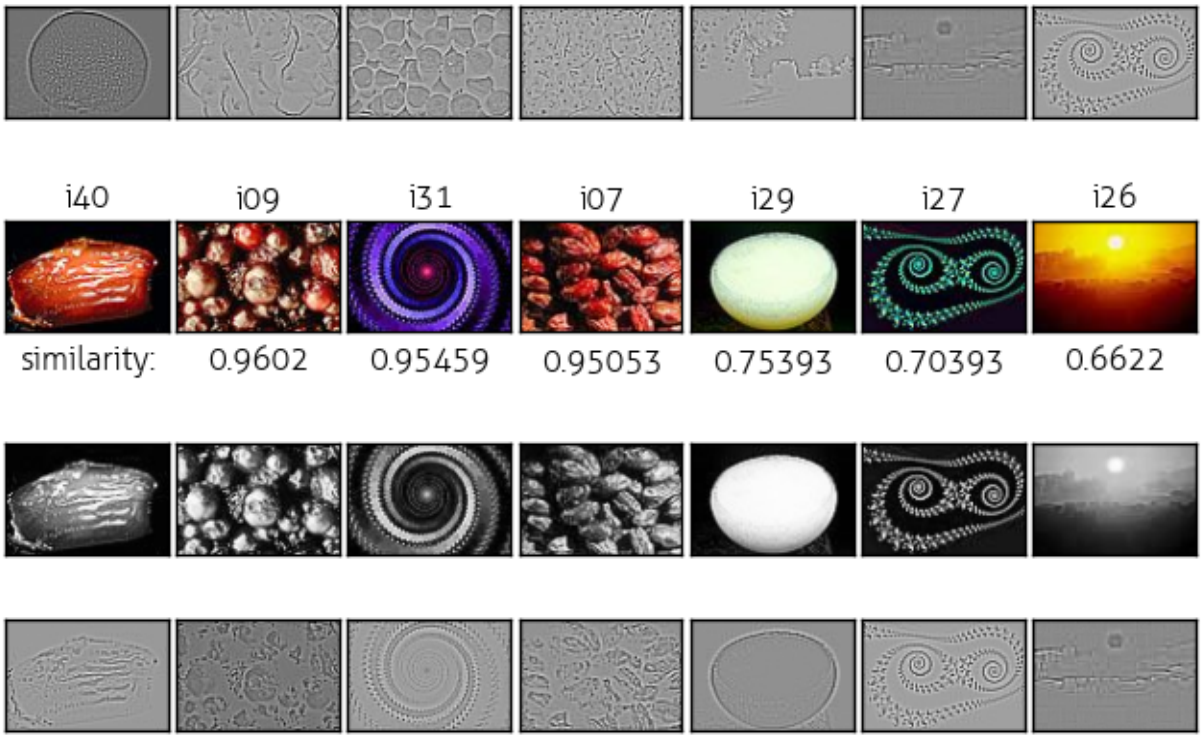

i27 i28 i33 i24 i12 i15 i26
similarity: 0.79721 0.779 0.75046 0.61148 0.60841 0.57135
i28 i24 i33 i03 i12 i15 i26
similarity: 0.92912 0.91135 0.8976 0.7219 0.71636 0.6424


i29 i30 i35 i34 i08 i01 i27
similarity: 0.93415 0.89522 0.88852 0.71744 0.70451 0.63887
i30 i35 i29 i22 i08 i01 i27
similarity: 0.94715 0.93415 0.9274 0.74302 0.72926 0.65003
i31 i17 i40 i09 i29 i27 i26
similarity: 0.95933 0.95459 0.95336 0.76103 0.69963 0.66791


i32 i21 i33 i34 i15 i27 i26
similarity: 0.92604 0.90969 0.90641 0.74979 0.72364 0.70965
i33 i28 i32 i24 i12 i15 i26
similarity: 0.91135 0.90969 0.89545 0.71577 0.70808 0.67577
i34 i21 i37 i30 i08 i01 i27
similarity: 0.93924 0.91992 0.91872 0.76331 0.74455 0.69299


i35 i22 i37 i30 i03 i01 i27
similarity: 0.9711 0.95034 0.94715 0.73127 0.71787 0.64061
i36 i18 i06 i20 i01 i26 i27
similarity: 0.96924 0.94868 0.93813 0.78244 0.72359 0.654


i37 i20 i35 i22 i03 i01 i27
similarity: 0.96396 0.95034 0.94149 0.7531 0.73728 0.65057
i38 i19 i17 i23 i29 i27 i26
similarity: 0.97014 0.96163 0.9494 0.78877 0.69956 0.68567
i39 i05 i16 i11 i25 i26 i27
similarity: 0.96654 0.95475 0.94975 0.77117 0.70834 0.65105


i40
i09
i31
i07
i29
i27
i26
similarity:
0.9602
0.95459
0.95053
0.75393
0.70393
0.6622