Step 0 - Hardware, Software, and Program Details

The bulk of this assignment was completed using **MATLAB**, with applications such as **Excel**, **Word**, **Photoshop**, and Apple's **Preview** for minor tasks. I'm working on a 2014 Macbook Pro with 8GB memory and a 2.4 GHz Intel Core i5 processor.

The MATLAB program runs rom the **HW2.m** script and uses functions and constants from other files. In this script, the four sections are labeled with **%% Step X** headers, where X is the step number.

The **Step 0** section of the HW2.m script loads all 40 images to be processed. They are used so frequently that I found it worthwhile keeping them in memory rather than loading them every time they are needed.

It's also worth noting that I developed this system against four user studies - three of them friends, one of them from myself. Given that I might have some bias and may have subconsciously submitted responses that were in-line with how the system was designed to work, I discounted the value of the system's score against them. Oddly enough, the system typically scored better against the other users than it did against me (a promising sign). All the same, throughout this paper, my responses are referred to as **user two** (or ClusterTwo / SurveyTwo).

These user studies ultimately became incredibly useful in tweaking features like the number of bins for the RGB histograms and the pixel threshold that distinguished the background from the foreground. On my first pass through the development, I arguably spent too much time playing with different settings to try to produce output that I thought was reasonable. With no real metric to compare outcomes against, this could change by the day. Once I had all of the user feedback loaded up and a mechanism for scoring the results, it was much easier to consistently spot improvements in the output. Of course, all of this comes down to how I chose to score the results, but was definitely an improvement over me eyeballing it.

Given the usefulness of the user studies throughout the development of the system, we'll proceed in the following way:

- **Step 0.5 - User Results:** We can arguably considered this Step 4a if we were following the assignment's step numbers. We'll show the user results, discuss them, and talk about how we'll score the system.
- **Step 1 - Gross Color Matching**
- **Step 2 - Gross Texture Matching**
- **Step 3 - Combining Similarities and Clustering**
- **Step 4 - Discussion on Results**
- **References**
- **Appendix (the MATLAB code)**

Note: *As I progress through the HW2.m script, the names of function being described will be put in* ***bold*** *in the headers.*

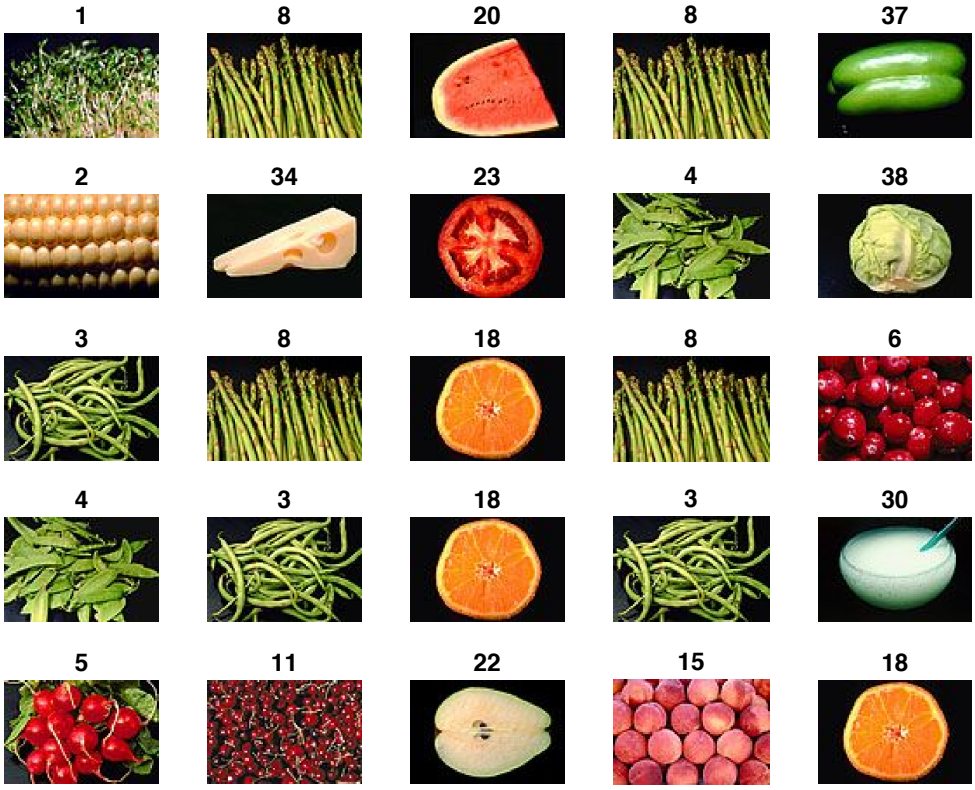
Step 0.5 - User Results (say, Step 4a)

It's worth starting this paper with the user results as they are often used to determine certain parameters in the following steps. So, let's begin by looking at the user matched similar and dissimilar images on color and texture, how they clustered the images, and how we plan on scoring the system against this data. This will allow me to explain certain implementation choices in the following steps where the system does color matching, texture matching, and its version of clustering. Finally, in **Step 4 (or Step 4b)**, I'll give all of the results.

For the color/texture matching test, the output is as described in the assignment. The first image in each row is the one we're finding matches for. The second and third images are the most similar in color and least similar in color, respectively. The fourth and fifth images are the most similar in texture and least similar in texture, respectively. The groups for the clustering step will be made clear.

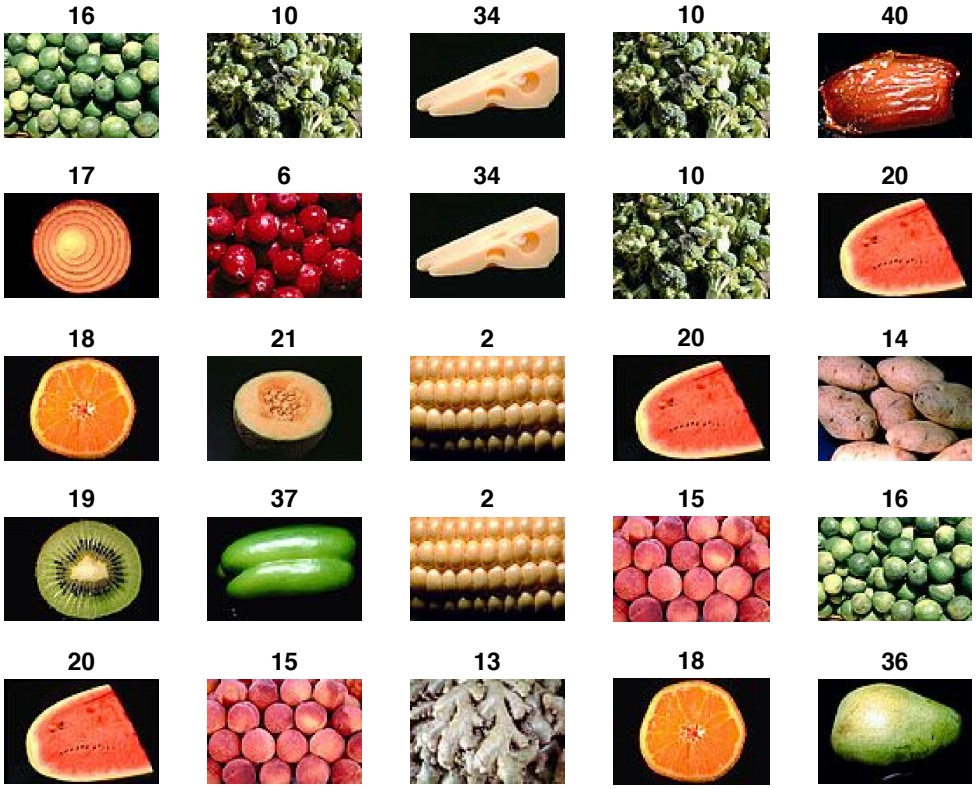
Again, I used data for four users. Users 1, 3, and 4 were friends, User 2 is myself.

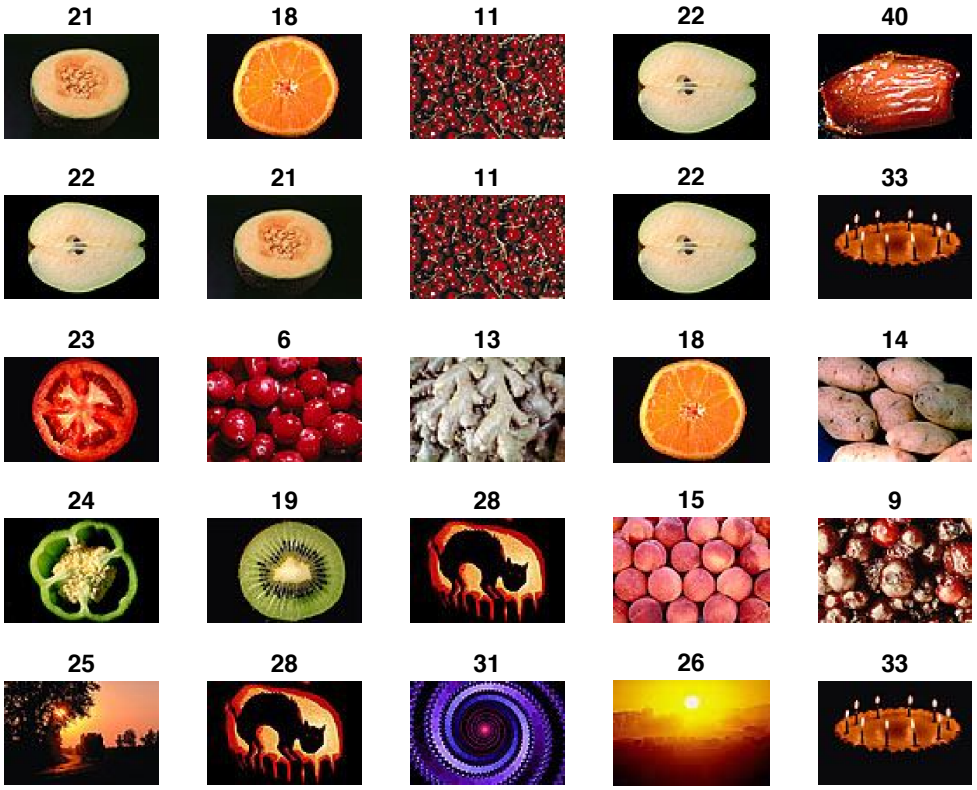
Similarity Matching - *User 1*

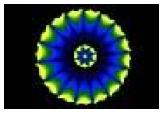
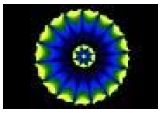
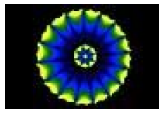


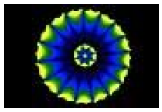
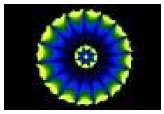
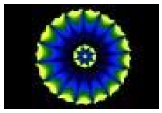
6	11	22	15	21
				
7	40	38	40	29
				
8	3	5	3	18
				
9	11	14	17	23
				
10	16	13	16	23
				

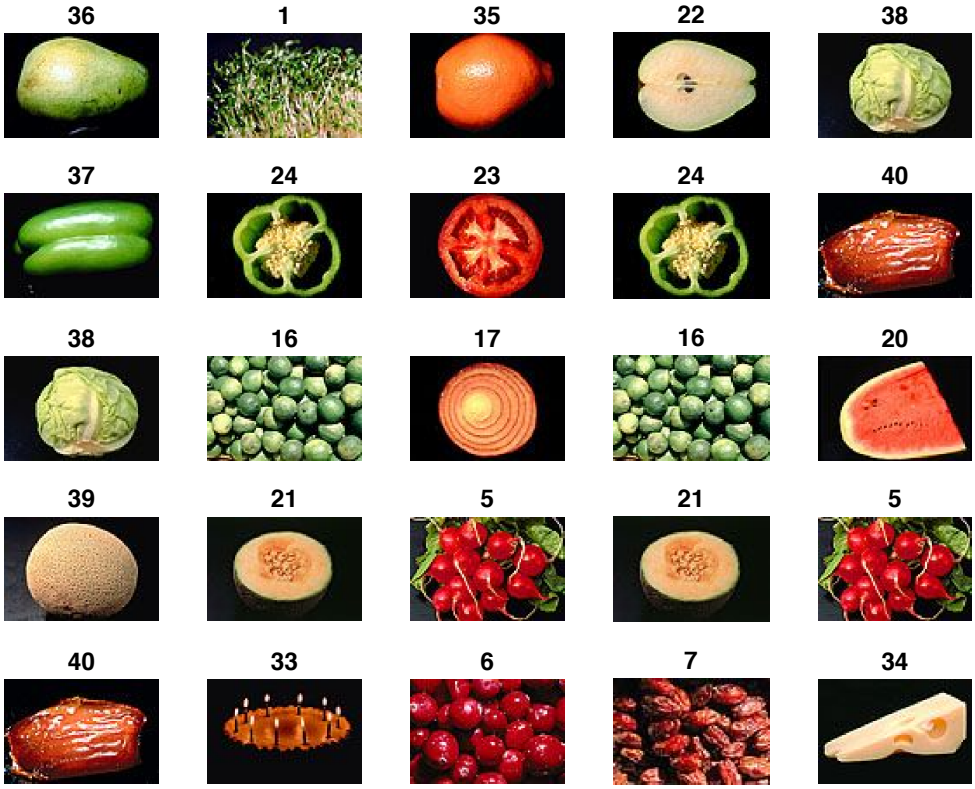
11	6	13	6	36
				
12	33	13	13	20
				
13	29	11	12	18
				
14	15	13	13	18
15	23	34	18	14


16 10 34 10 40
17 6 34 10 20
18 21 2 20 14
19 37 2 15 16
20 15 13 18 36

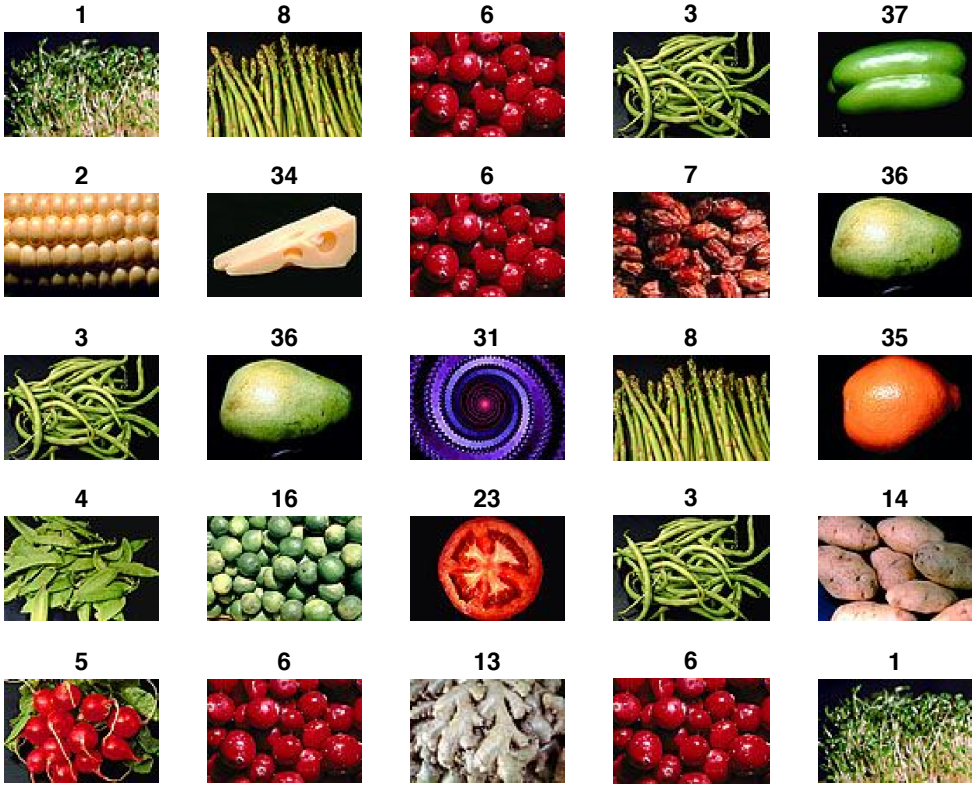

21 18 11 22 40
22 21 11 22 33
23 6 13 18 14
24 19 28 15 9
25 28 31 26 33

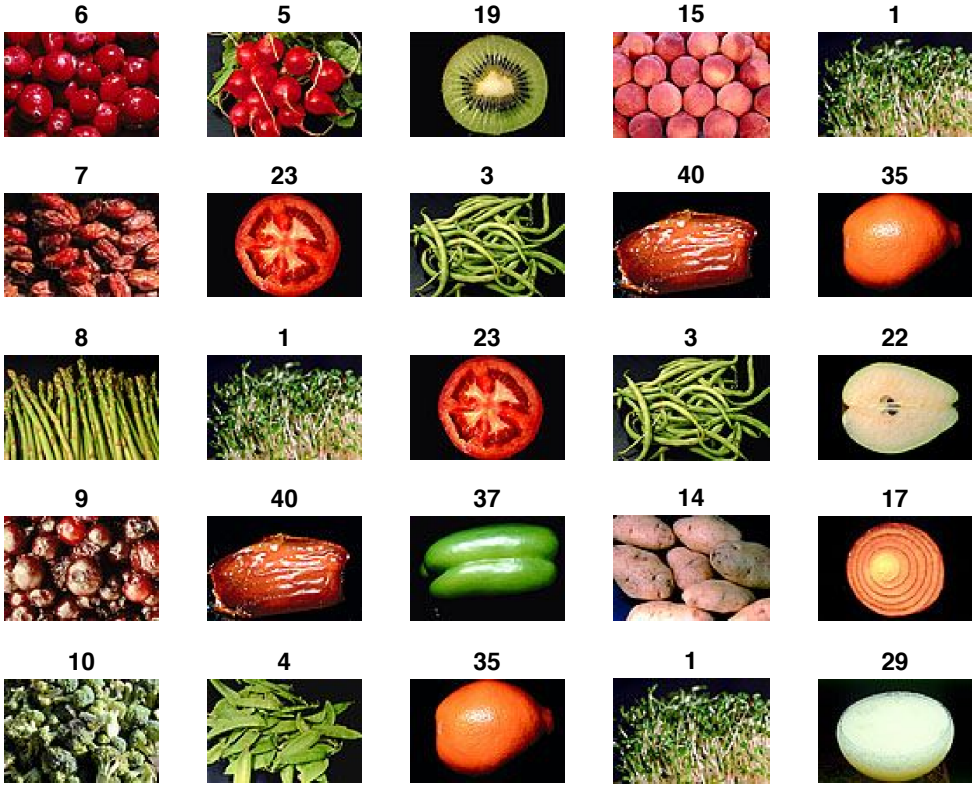
26	25	31	25	31
				
27	32	31	31	33
				
28	33	32	33	32
				
29	22	5	30	9
				
30	1	11	29	39
				

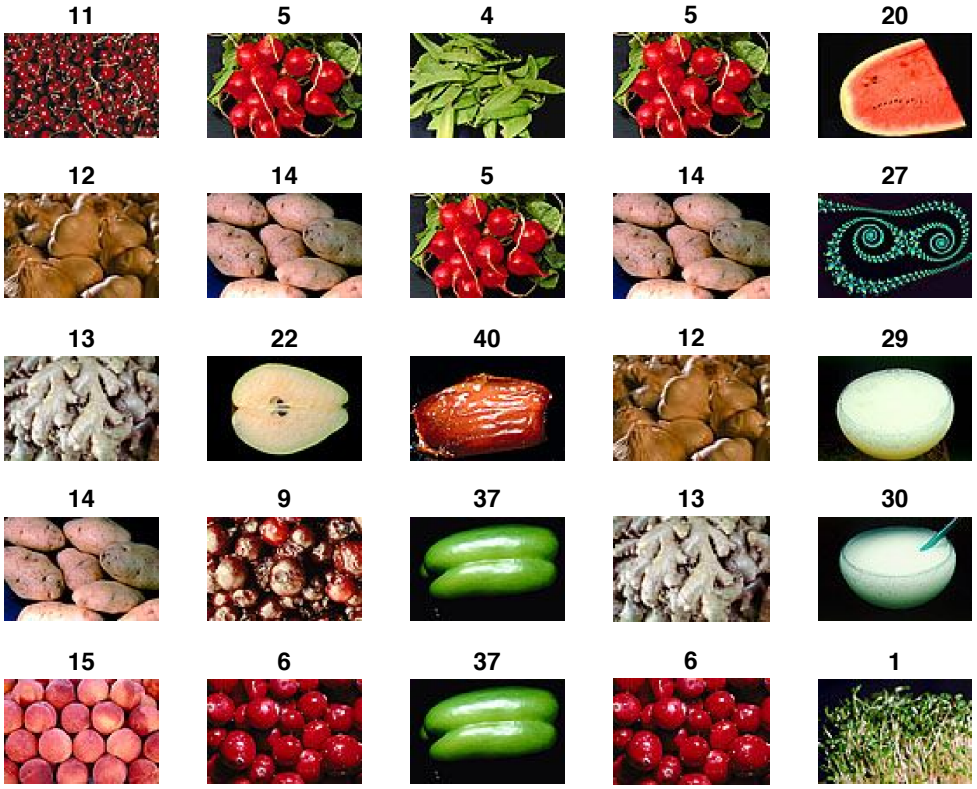
31	32	29	32	33
				
32	27	28	31	13
				
33	28	31	34	14
				
34	22	23	33	35
35	18	11	39	37

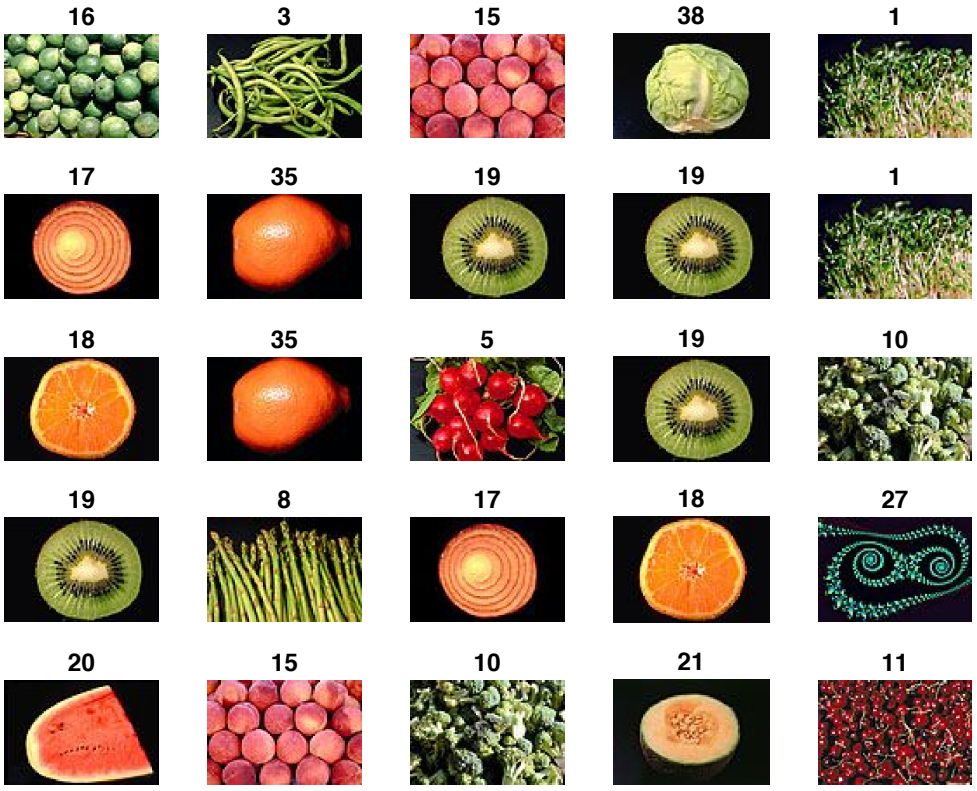

36
1
35
22
38
37
24
23
24
40
38
16
17
16
20
39
21
5
21
5
40
33
6
7
34

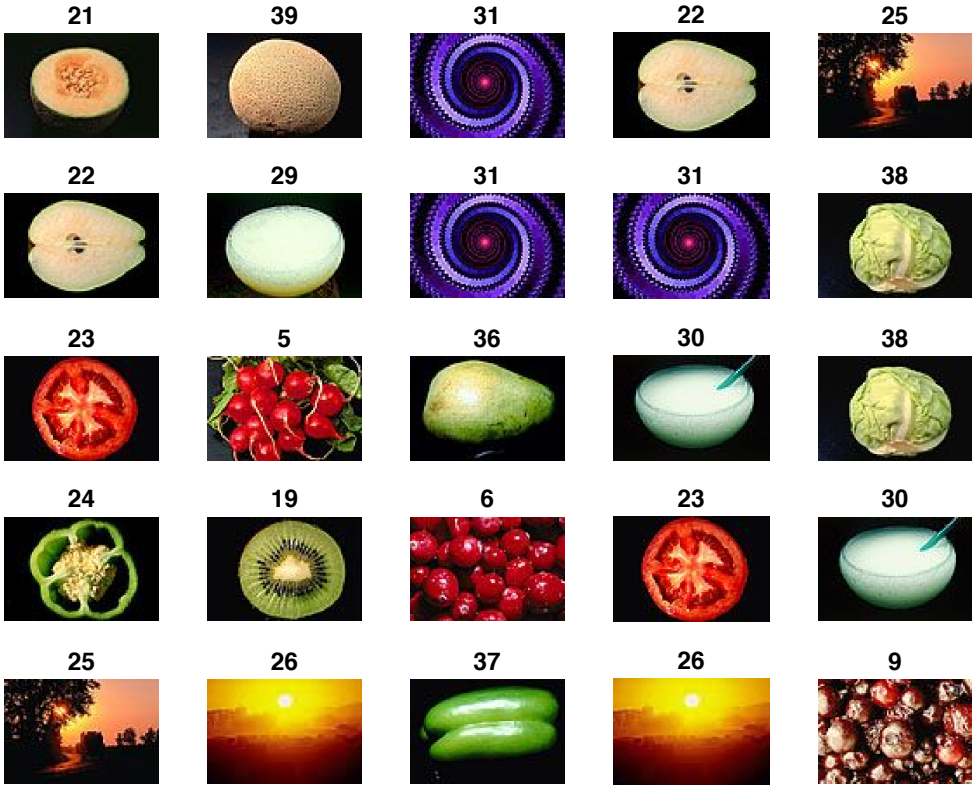
Similarity Matching - User 2

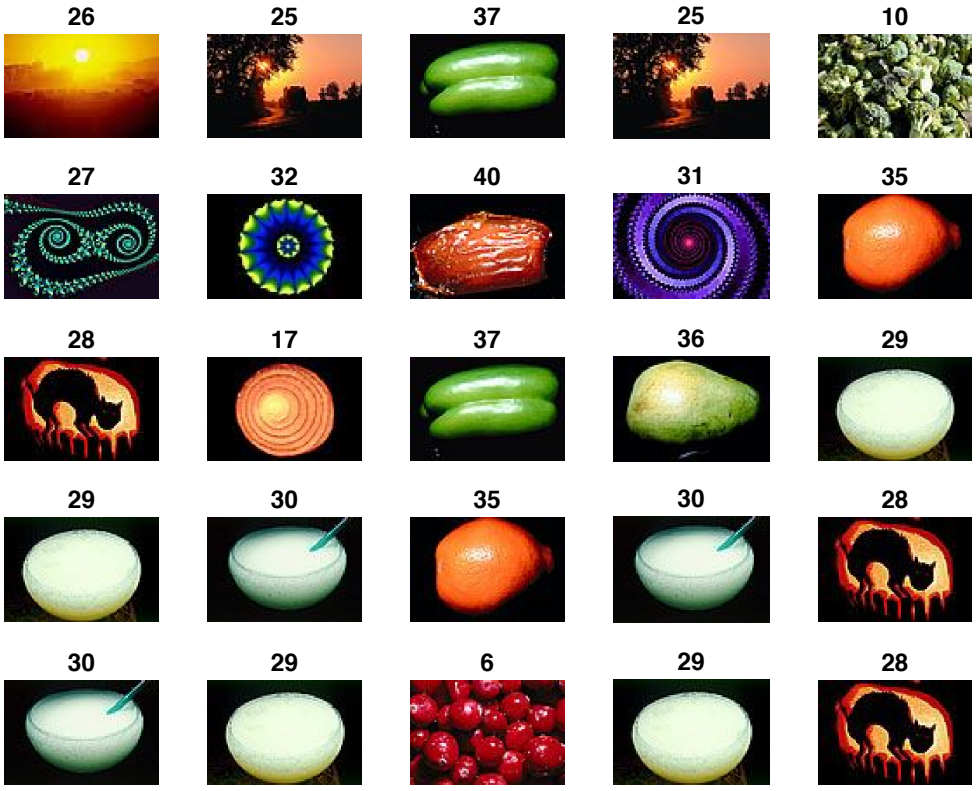


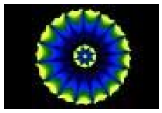
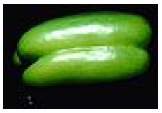



11
5
4
5
20
12
14
5
14
27
13
22
40
12
29
14
9
37
13
30
15
6
37
6
1

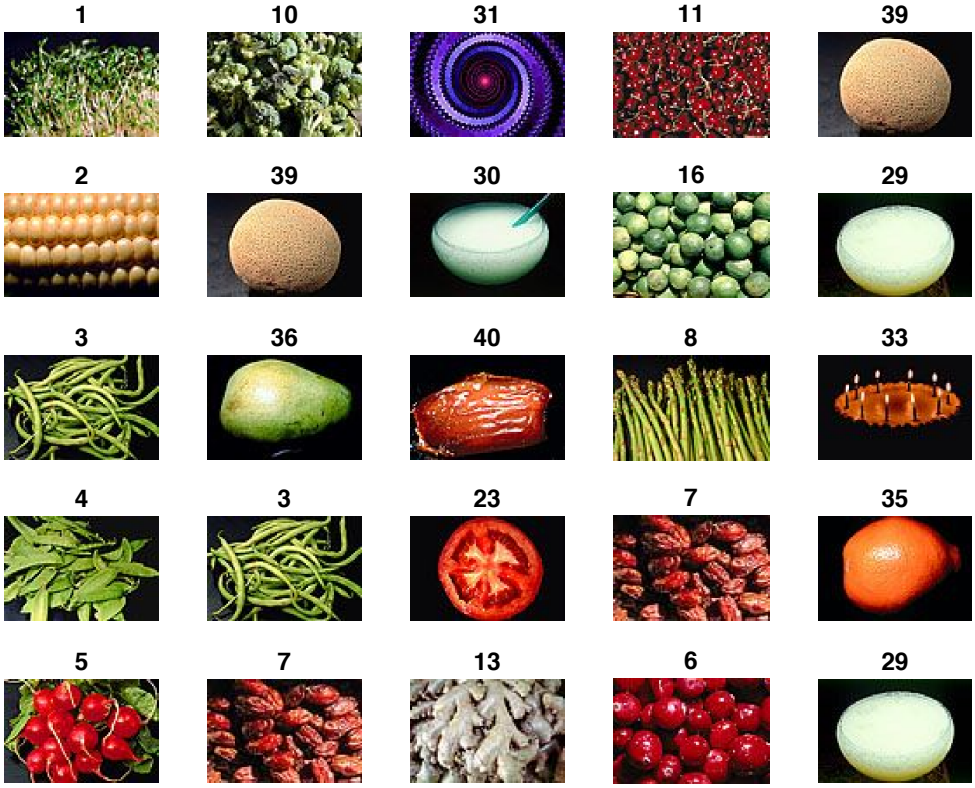

16
3
15
38
1
17
35
19
19
1
18
35
5
19
10
19
8
17
18
27
20
15
10
21
11

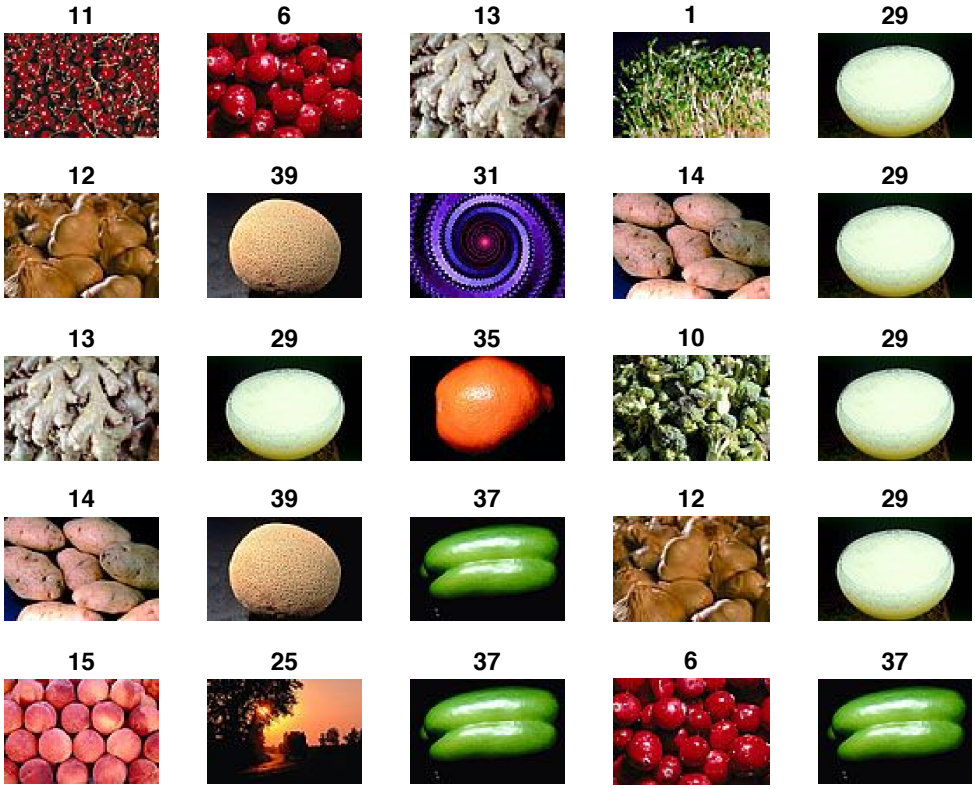

21 39 31 22 25
22 29 31 31 38
23 5 36 30 38
24 19 6 23 30
25 26 37 26 9

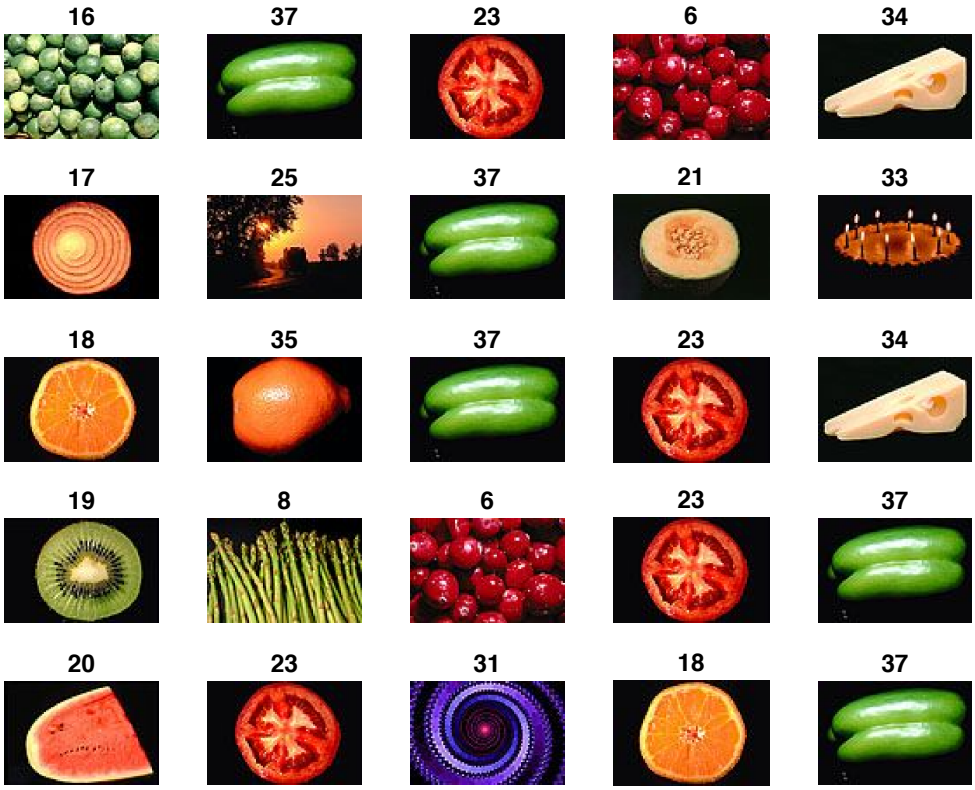

26 25 37 25 10
27 32 40 31 35
28 17 37 36 29
29 30 35 30 28
30 29 6 29 28

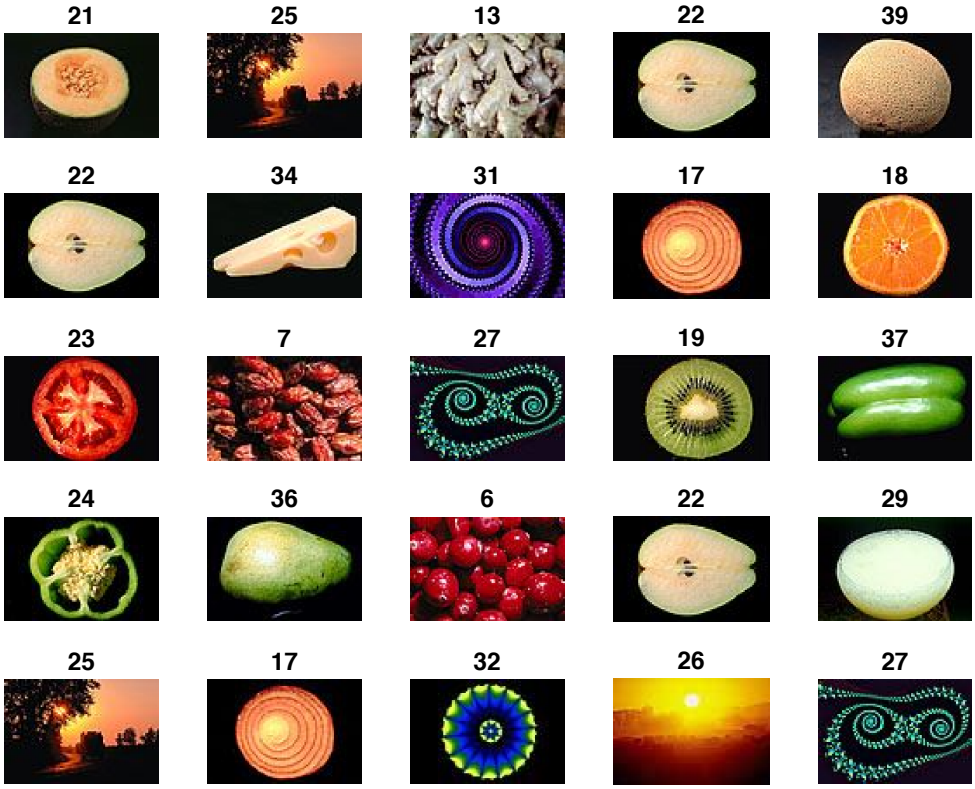
31	7	22	27	9
				
32	27	6	33	27
				
33	28	37	34	9
				
34	39	37	36	9
				
35	18	36	36	10
				
36	3	35	37	9
				
37	4	15	36	27
				
38	36	6	16	20
				
39	34	5	27	1
40	11	4	7	27

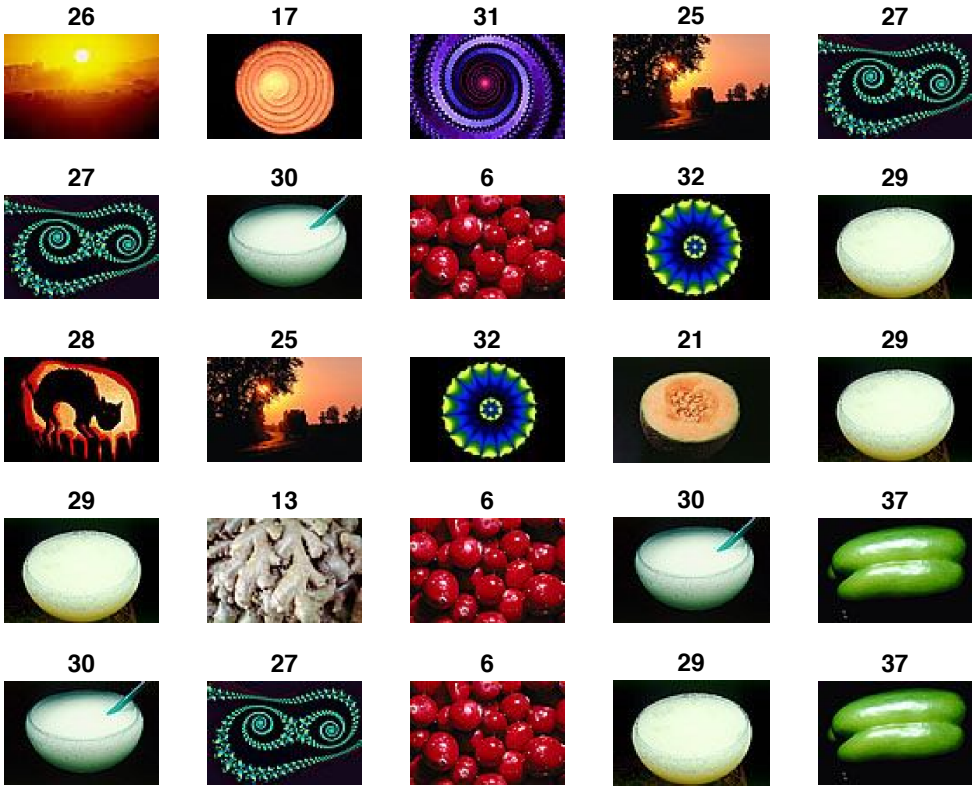
Similarity Matching - User 3

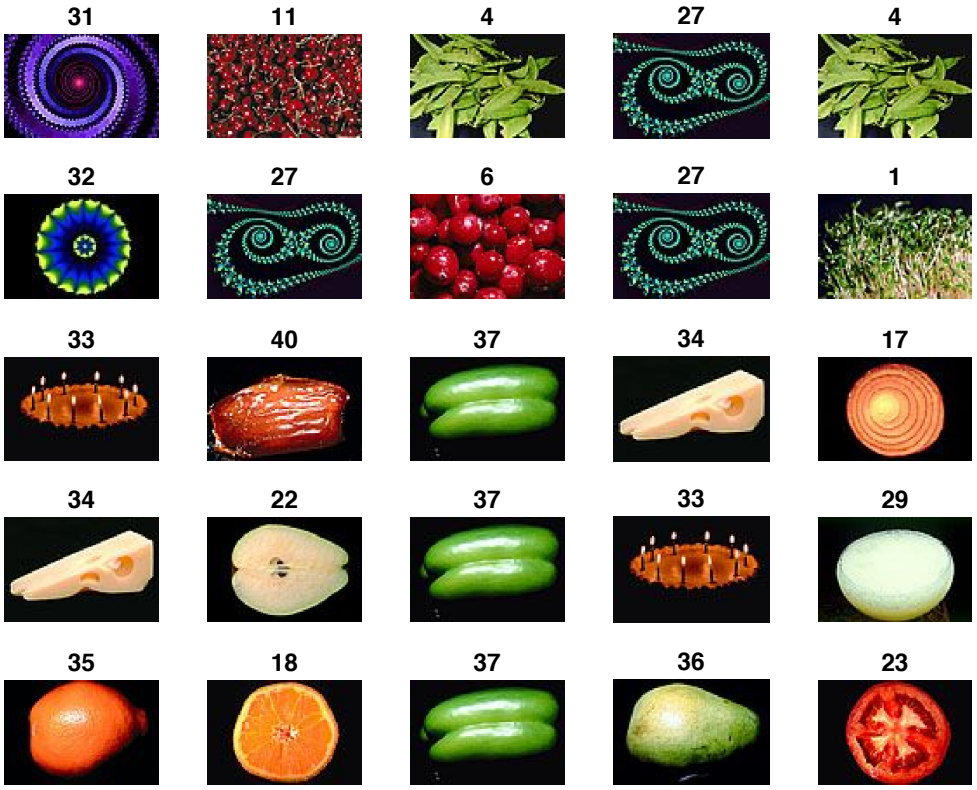


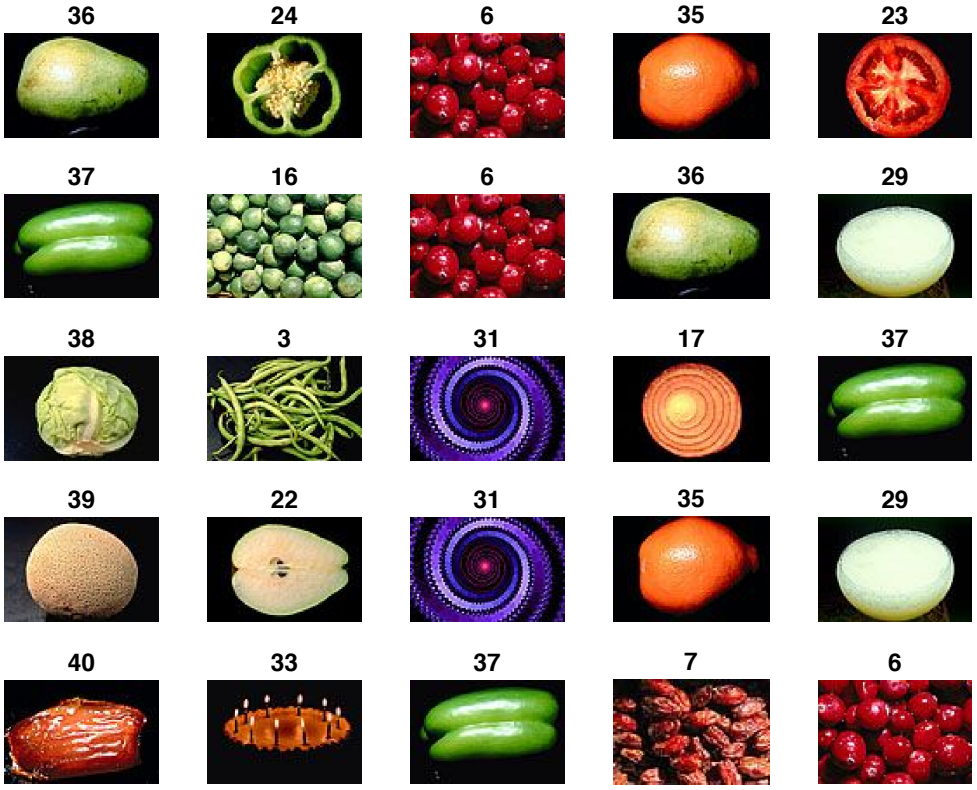

11
6
13
1
29
12
39
31
14
29
13
29
35
10
29
14
39
37
12
29
15
25
37
6
37


16
37
23
6
34
17
25
37
21
33
18
35
37
23
34
19
8
6
23
37
20
23
31
18
37

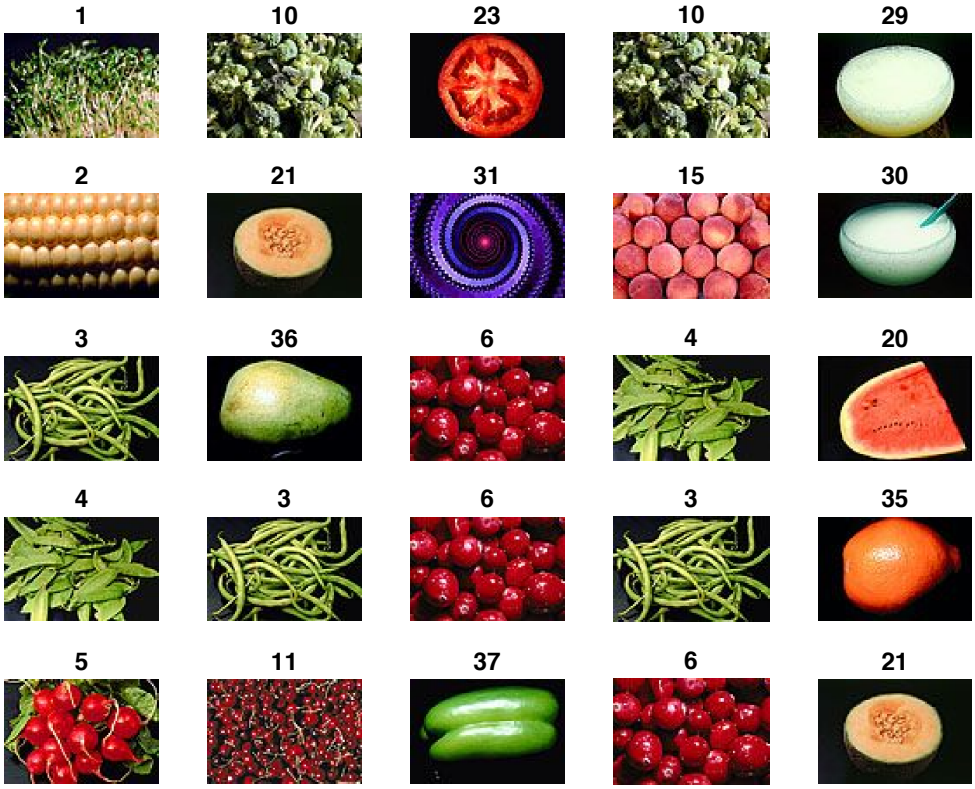

21 25 13 22 39
22 34 31 17 18
23 7 27 19 37
24 36 6 22 29
25 17 32 26 27

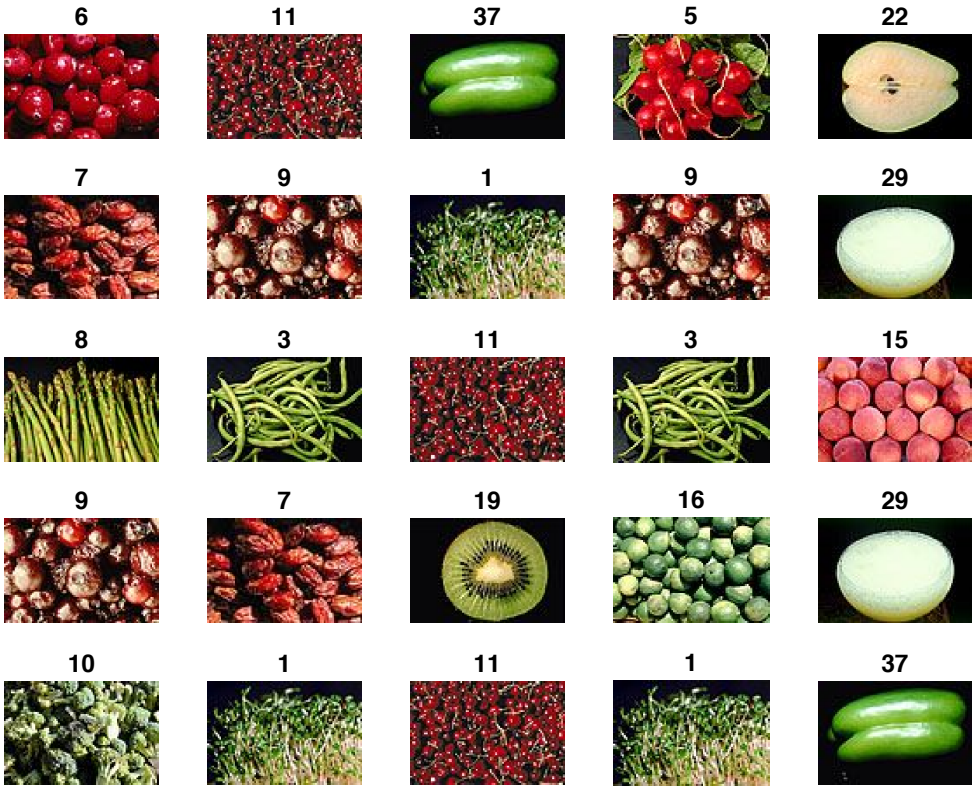

26 17 31 25 27
27 30 6 32 29
28 25 32 21 29
29 13 6 30 37
30 27 6 29 37

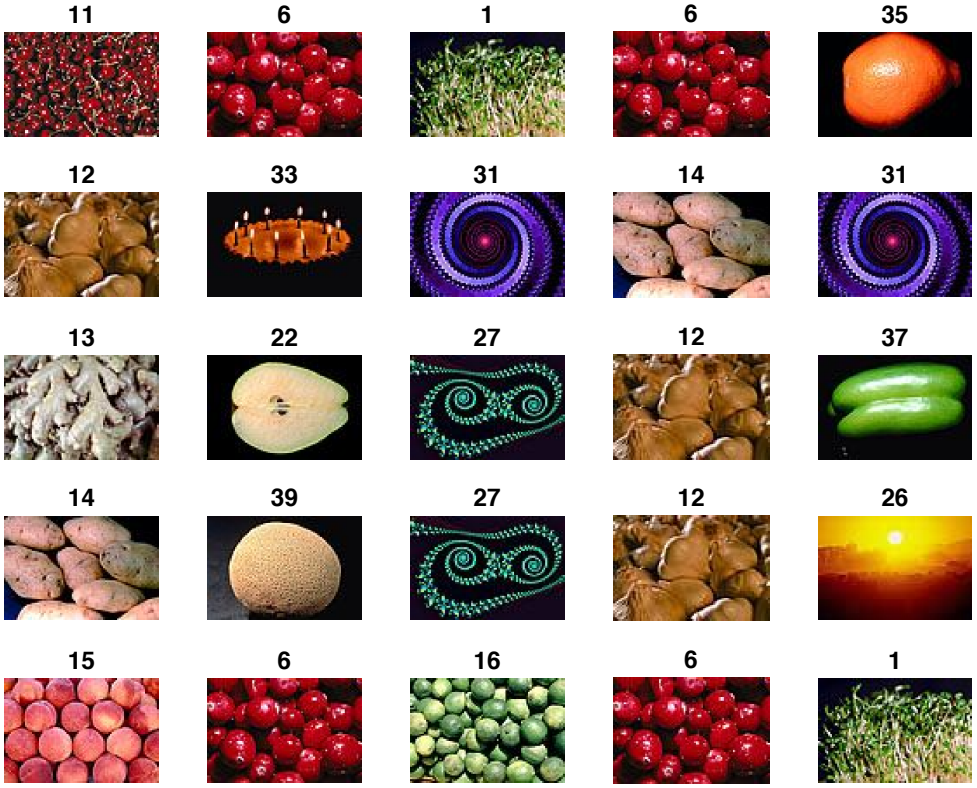

31
11
4
27
4
32
27
6
27
1
33
40
37
34
17
34
22
37
33
29
35
18
37
36
23

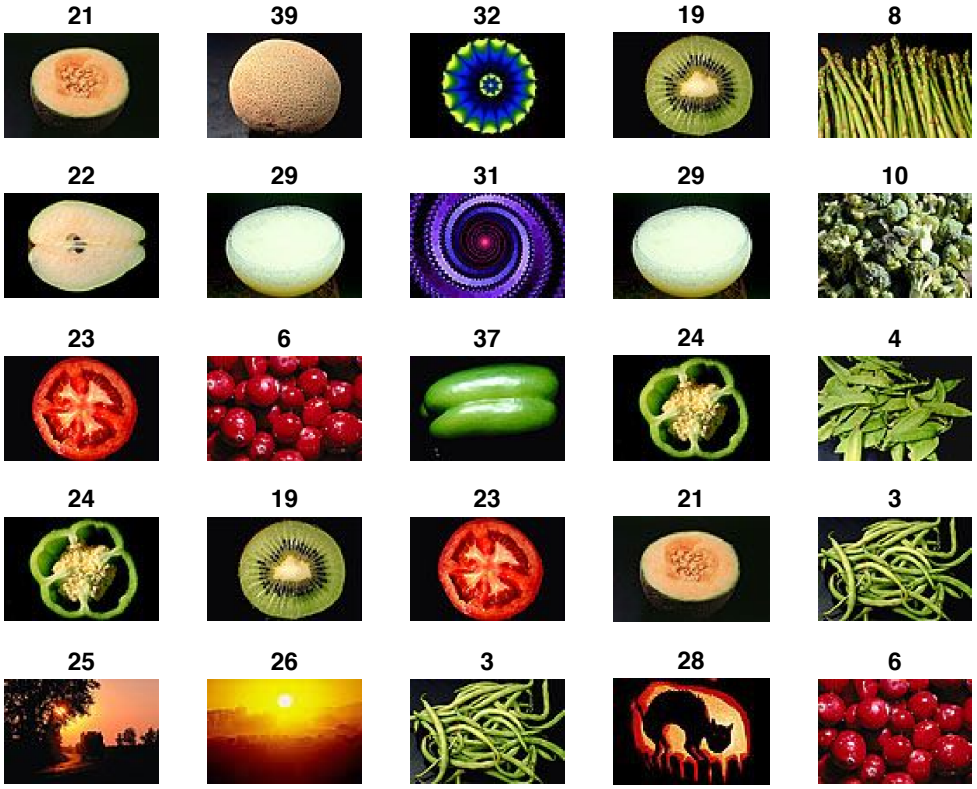

36
24
6
35
23
37
16
6
36
29
38
3
31
17
37
39
22
31
35
29
40
33
37
7
6

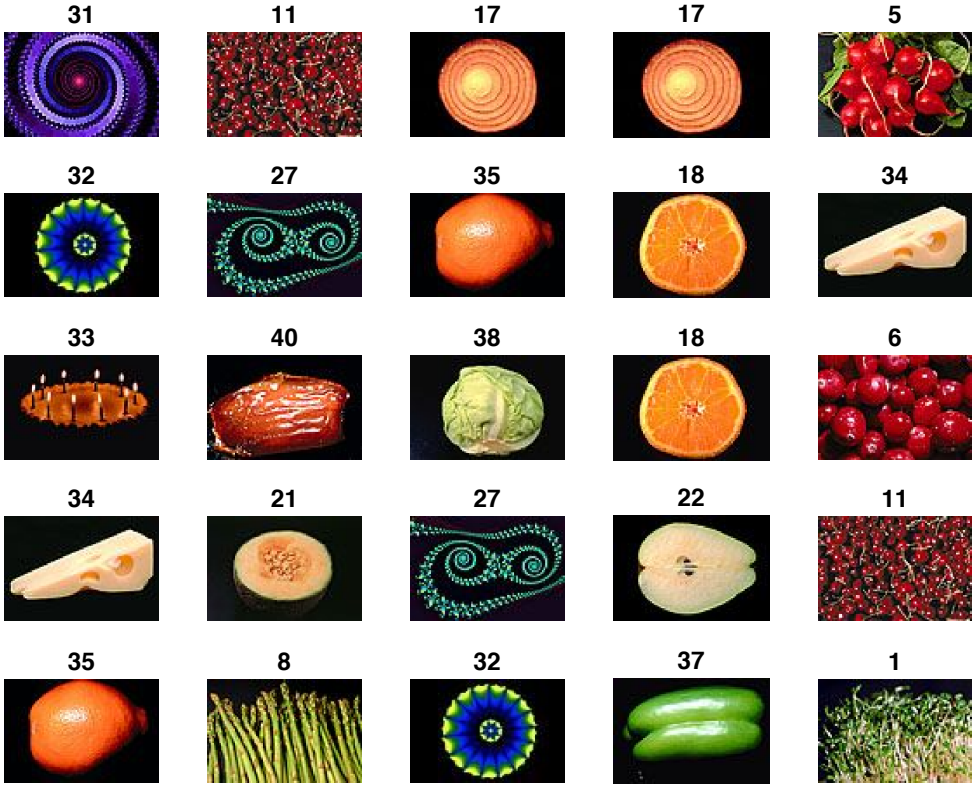
Similarity Matching - User 4

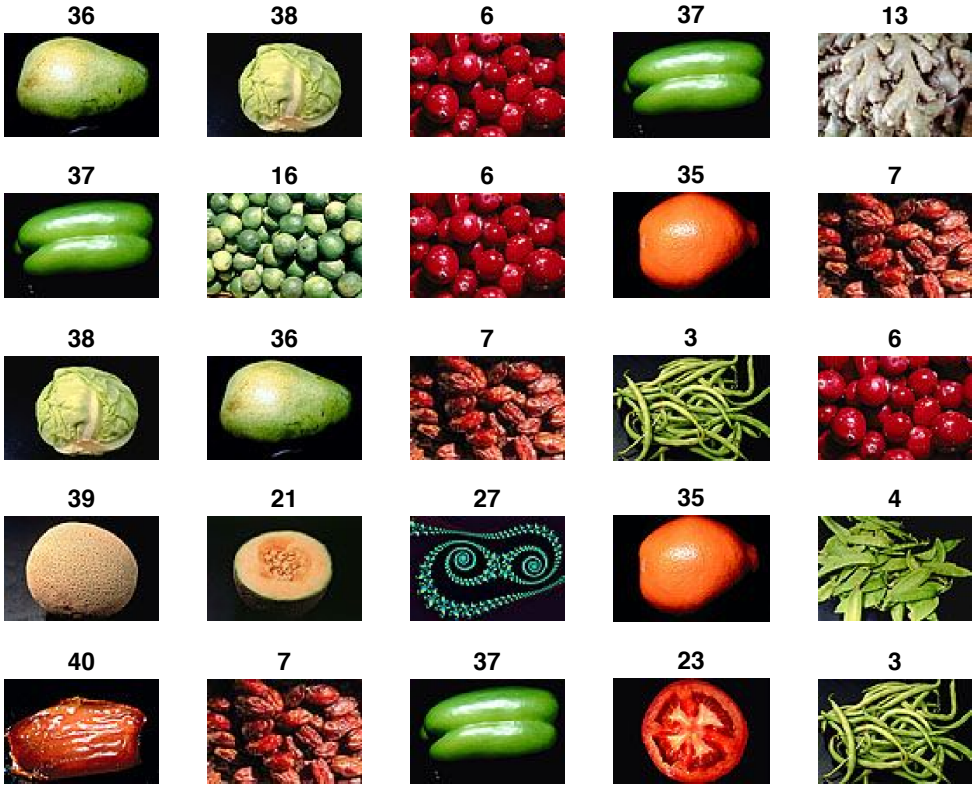





11
6
1
6
35
12
33
31
14
31
13
22
27
12
37
14
39
27
12
26
15
6
16
6
1
16
36
15
15
1
17
21
37
19
11
18
35
32
32
3
19
8
23
17
34
20
15
36
22
1


21 39 32 19 8
22 29 31 29 10
23 6 37 24 4
24 19 23 21 3
25 26 3 28 6
26 18 1 20 9
27 30 35 31 28
28 25 27 25 27
29 22 31 30 1
30 27 35 29 1


31
11
17
17
5
32
27
35
18
34
33
40
38
18
6
34
21
27
22
11
35
8
32
37
1


36
38
6
37
13
37
16
6
35
7
38
36
7
3
6
39
21
27
35
4
40
7
37
23
3

Clustering Results - User 1

Cluster 1

2 25 26

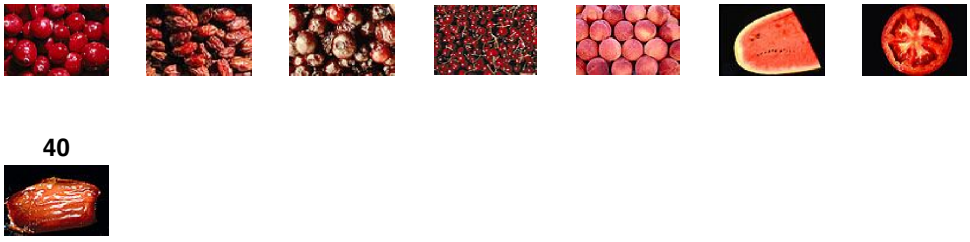
Cluster 2

18 21 35 39

Cluster 3

6 7 9 11 15 20 23

40



Cluster 4

27 29 30 32

Cluster 5

1 3 4 8 10 22 24

36 37

Cluster 6

5 

12 

13 

14 

16 

34 

Cluster 7

17 

19 

28 

31 

33 

38 

Cluster Results - User 2

Cluster 1

29 

30 

Cluster 2

2 

18 

21 

25 

26 

34 

35 

39 

Cluster 3

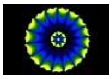
28 

33 

Cluster 4

27 

31 

32 

Cluster 5

1 

3 

4 

8 

10 

16 

19 

22 

24 

36 

37 

38 

Cluster 6

5 

6 

7 

11 

15 

17 

20 

23 

40 

Cluster 7

9 

12 

13 

14 

Cluster Results - User 3

Cluster 1

13 29 30

Cluster 2

27 31 32

Cluster 3

9 12 14 28 33

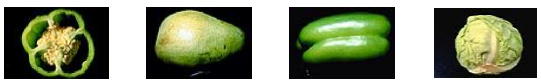
Cluster 4

2 22 34 39

Cluster 5

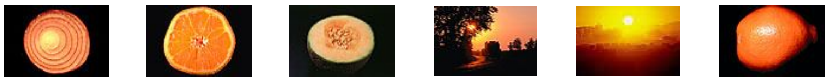
1 3 4 8 10 16 19

24 36 37 38



Cluster 6

17 18 21 25 26 35



Cluster 7

5

6

7

11

15

20

23

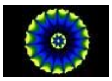
40

Cluster Results - User 4

Cluster 1

20 
23 
25 
28 
34 
40 

Cluster 2

27 
31 
32 

Cluster 3

1 
3 
4 
8 
10 
16 
24 

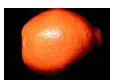
Cluster 4

2 
12 
13 
14 

Cluster 5

5 
6 
7 
9 
11 
15 

Cluster 6

17 
18 
21 
26 
33 
35 
39 

Cluster 7

Scoring the System

Now that we have the user output, we can talk about how we'll score the system.

Scoring - Similarity Matches

The system will output an image and 6 matches for the similarity matching steps. The six images are in the following order: most similar, 2nd most similar, 3rd most similar, 3rd most dissimilar, 2nd most similar, most dissimilar. For each image, it will create this output when matching both color and texture. The hope is that the system will output matches that are similar to the user studies. In other words, if the user said image X is the most similar in color to image Y, the hope is that the system will have image X is one of the three "most similar" positions of the color matches for Y.

Since the users are simply saying that an image is most similar or most dissimilar, it seems reasonable to think that the system should be rewarded for matching the user output with images in the "most similar" or "most dissimilar" columns (rather than 2nd or 3rd most). This led me to create the following scoring mechanism:

- The system gets 3 points if it matches the user output in its "most similar/dissimilar" column
- The system gets 2 points if it matches the user output in its "2nd most similar/dissimilar" column
- The system gets 1 point if it matches the user output in its "1st most similar/dissimilar" column
- The system gets 0 points if it doesn't match

If the system matches perfectly with a user for all "most similar" results, it will get a perfect score of 120 (40 * 3). The same is true for "most dissimilar" results. This leads to a combined score of 240. This score is calculated for both color and texture separately.

Possible Score: [0, 240]

Scoring - Clustering Matches

To score itself against user clusters, the system uses the Rand index. Essentially, it looks at pairs of elements within the set being clustered. If these two elements appear in the same cluster in both clusterings, the two partitions are believed to be more similar. Similarly, if the two elements are both separated into different clusters in each clustering, the two partitions are believed to be more similar as well. In contrast, if any two elements are grouped into the same cluster in one partition, but are in separate clusters in the second partition, the sets are believed to be less similar. These comparisons are made for all pairs of elements in the original set.

In writing the algorithm that performed this calculation, I followed the definition defined here: *http://en.wikipedia.org/wiki/Rand_index*

Possible Score: [0, 1]

With these scoring mechanisms in mind, we can now proceed to the systems processing and analysis of the images.

Step 1 - Gross Color Matching

We begin with how the system compares images to each other based on color. This text refers to **%% Step 1** section of the code in **HW2.m**.

Getting the Normalized Color Histogram - **getNormalizedColorHistogram(img, sgmts)**

To enable color comparisons between images, I first reduced each image into a normalized, 3D histogram where the "bins" of this histogram corresponded to a certain range of RGB values. Specifically, these bins contained the percentage of "significant pixels" in the image that fell within their range.

The bins were created by dividing up the 255 channels that each color (red, green, blue) could be measured at. I ultimately chose to divide these channels up into 7 segments, which made each segment approximately 36 in length. Given there are three channels, this resulted in 343 bins (7*7*7) where each bin was ~ 36 x 36 x 36 in size.

As an example, we could say the first bin is the one that contains the number of pixels that have red, green, and blue values all between 0 and 36. In an image, these are pixels very dark pixels, ranging from pure black to mostly black.

Note: *7 segments per color channel was chosen was based on matching the systems output to the user output. These scores are the combined total for similar and dissimilar matches (out of 240). Details of the scoring system are found in* ***Step 0.5****. Overall scores for color matching at found in* ***Step 4.***

# Segs	3	4	5	6	7	8	9	10
User 1	44	41	35	36	40	34	34	35
User 2	45	62	66	70	68	71	74	71
User 3	90	98	95	97	102	95	101	102
User 4	76	92	81	82	89	84	89	89
Average	**63.75**	**73.25**	**69.25**	**71.25**	**74.75**	**71**	**74.5**	**74.25**

The histograms are initially built with the number of pixels that fall within a certain bin. In the function, this can be seen with `hist(rs, gs, bs) = hist(rs, gs, bs) + 1`. However, before doing the analysis, we want to exclude background pixels so that comparisons are primarily based on the primary content of the image. Fortunately, almost all of these images had a shade of black as their background, which made it possible to reasonably distinguish these background pixels from the content.

To get a feel for what RBG values these background pixels could take, I opened several images in Photoshop and sample pixels from the background. In general, I found that each of the red, green, and blue values in a background pixel had values less than 35. This corresponded well to the bin sizes I used for my histogram as pixels that fell in the first bin [RGB values from

(0,0,0) to (36,36,36)] could be ignored when doing image comparisons. In other words, the pixels that fell outside of this bin, which I'll call the "background bin," were considered significant and likely apart of the main content of the image.

However, since every image can have a different number of background pixels, our histograms needed to be normalized in order to do any fair comparison. Specifically, we wanted to calculate the percentage of an image that fell into the rest of the color bins after excluding the background pixels. To do this, I kept track of the number of significant pixels as the algorithm iterated through all of the image pixels. If a pixel was classified to a bin besides the background bin, the number of pixels in the bin was incremented and this count of significant pixels was incremented. If a pixel was classified to the background bin, we do nothing and move to the next iteration.

Once each pixel of the image had been classified, we could take the corresponding 3D histogram and divide each bin by the number of significant pixels. The resulting values in the bins were the percentage of pixels (between [0, 1]) of the primary content (non-background) of the image that were classified to that bin. The background bin always had a value of 0.

With the normalized 3D histograms at hand, I then converted them into a (1 x 343) row vector using MATLAB's **reshape()** function.

Performing the Normalized L1 Comparison - **l1Compare(vec1, vec2)**

With the normalized color histogram vectors for each image, pairwise comparisons could be made to establish similarities.

The method for this is straightforward. Given two normalized vectors of the same dimension, one from the other and take the absolute value of each element of the resulting vector. Furthermore, to normalize this resulting vector, divide each bucket by 2 since the sum of all of the values in each of the original vectors was 1.

Vectors that were similar to each other will have a resulting vector where the sum of its values will be close to 0. Vectors that are dissimilar (with weights on different bins) will have a resulting vector with the sum of its values closer to 1. To align with the assignment instructions and a more intuitive result, the similarity output equals 1 minus the sum of the elements in this resulting vector. In other words, we output a value that is close to 1 if the two image color vectors are similar, or close to 0 if they are dissimilar.

The results of these comparisons are stored in a N x N matrix, where N is the number of images (40), that is symmetric across the top left to bottom right diagonal. Note, extra work wasn't done to create this symmetry as the resulting comparison between image i and image j where 1 <= i <= 40 and 1 <= j <= 40 was put in index (i, j) and (j, i) of the comparison matrix. In addition, all comparison results in indexes (n, m) where n == m has a value of 1 since the vectors fed into the function were identical (from the same image).

Getting the Similarity Results - **getSimilarityResults(comps)**

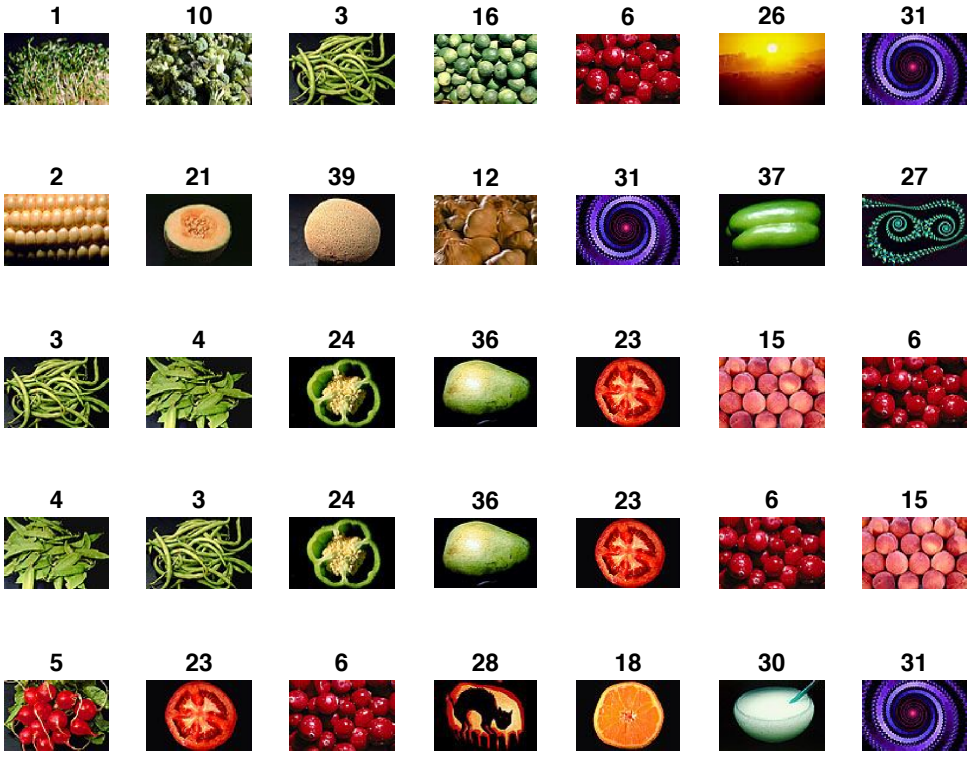
Now that we have this 40 x 40 matrix containing similarity values between [0, 1], we can find the top three most similar and top three most dissimilar matches for each image.

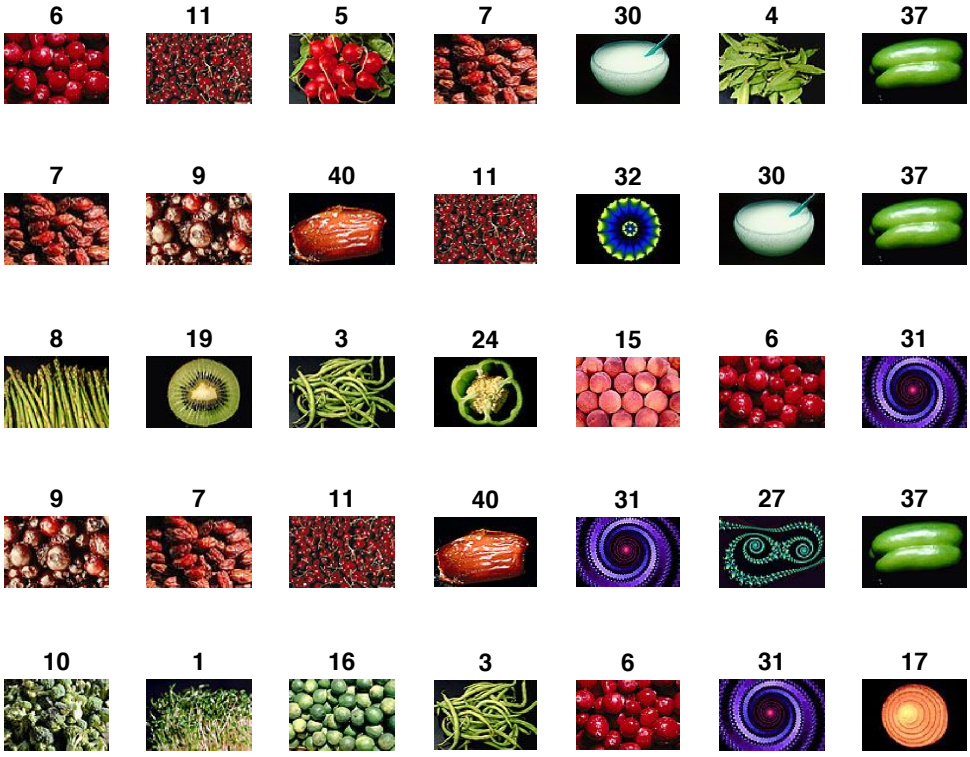
Although you could read it both ways, we'll say that each row **n** in this matrix contains 40 column row vector that contains the similarity value of comparing image **n** to image **m** where **m** is a column number.

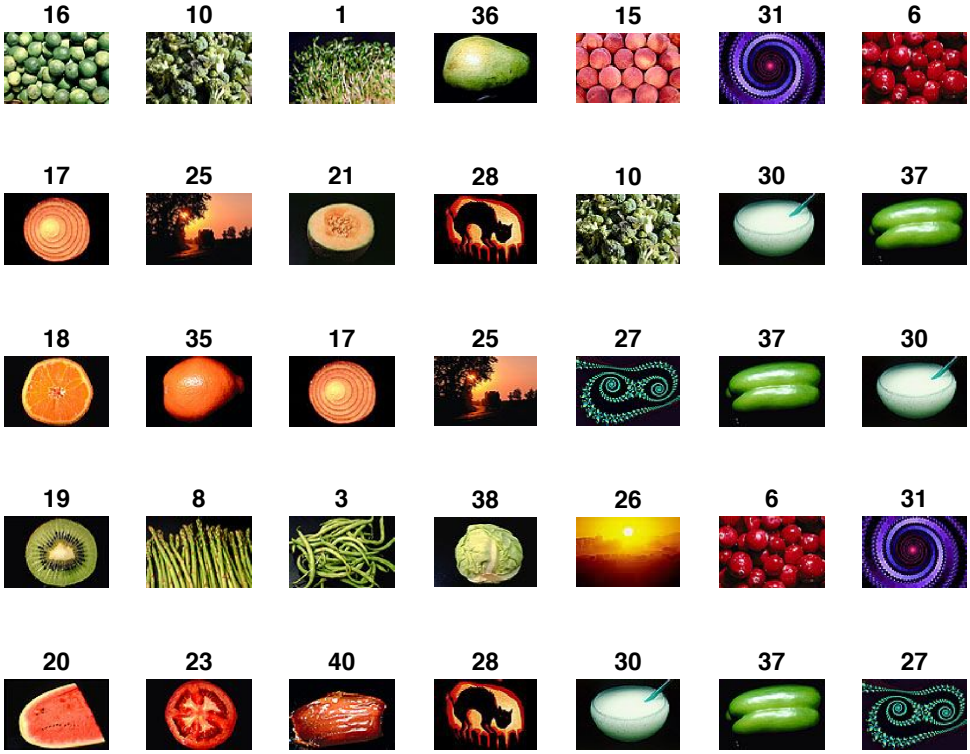
To find the top and bottom three matches, iterate **n** from 1 to 40 and grab each of these row vectors with the similarities one at a time. These row vectors are then appended to the bottom of another row vector from 1 to 40 that represents the image names. This results in a 2 x 40 matrix. The columns of this matrix is then sorted by the second row in increasing order. This results in a 2 x 40 matrix where the second row is sorted in increasing order and the first row contains the image number that **n** was compared with to achieve that score.

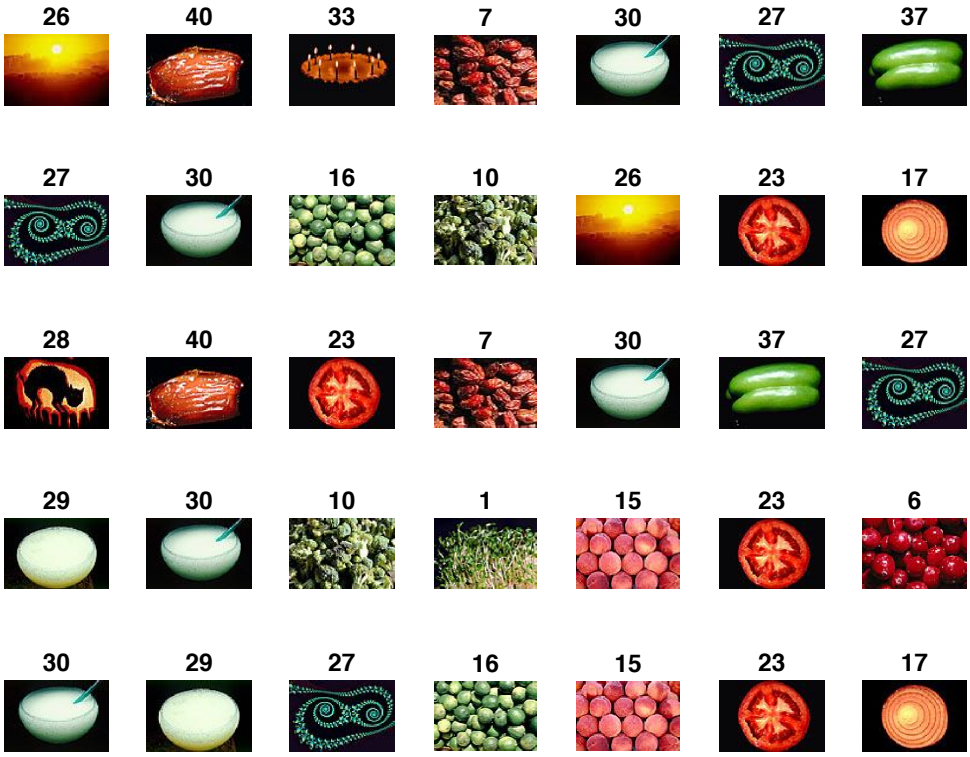
With this sorted matrix, it's easy to grab the images that are most similar to **n**. Assuming a total number of images **N**, (1,N-1) will be the most similar, (1,N-2)will be the second most similar, and (1,N-3) will be the third most similar ((1,N) would be **n** itself). The most dissimilar is also easy to locate. (1,1) is the most dissimilar, (1,2) is the second most dissimilar, and (1,3) is the third most dissimilar.

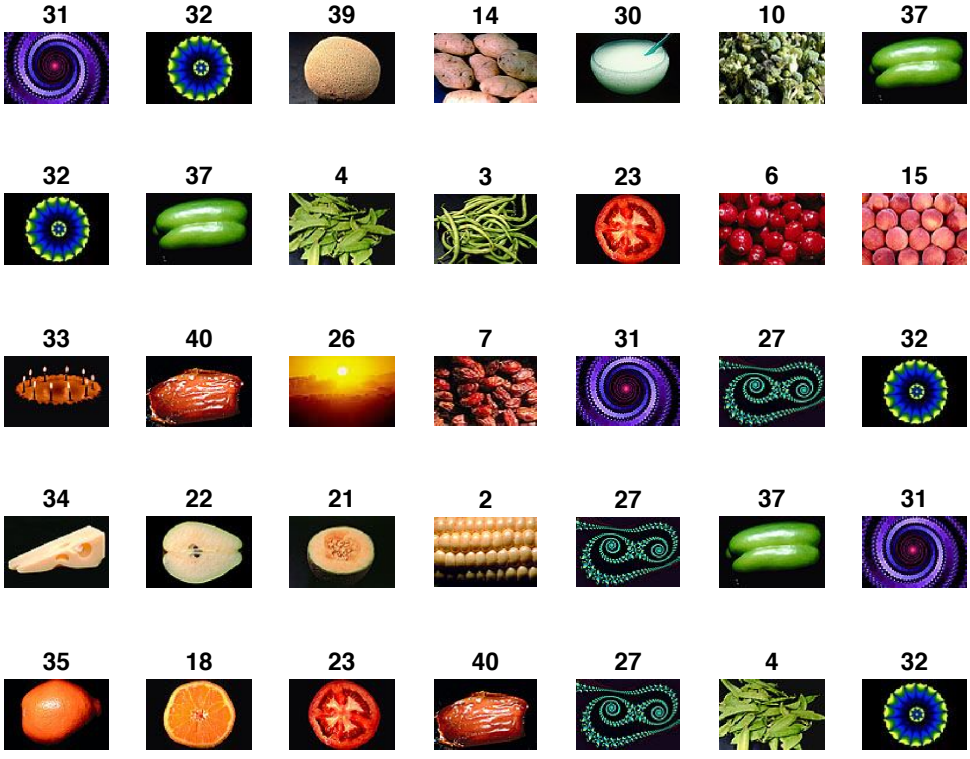
After doing this for each row, we'll have a 40 x 7 matrix that represents the desired output. The first column is the image being compared to the rest, and the next six images are the order described by the assignment (most similar to most dissimilar). This results are shown below with image names above the thumbnail.

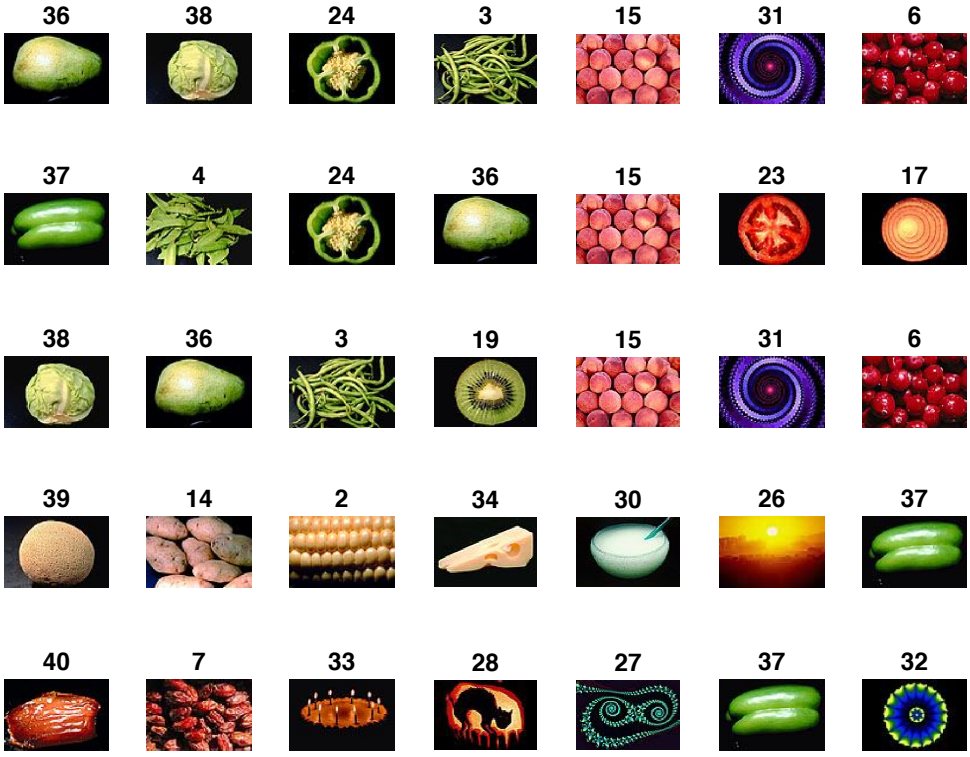



6 11 5 7 30 4 37
7 9 40 11 32 30 37
8 19 3 24 15 6 31
9 7 11 40 31 27 37
10 1 16 3 6 31 17
11 9 7 6 31 30 37
12 2 9 33 31 37 27
13 10 14 1 18 17 31
14 39 9 13 31 26 37
15 7 40 23 30 4 37


16 10 1 36 15 31 6
17 25 21 28 10 30 37
18 35 17 25 27 37 30
19 8 3 38 26 6 31
20 23 40 28 30 37 27
21 2 34 17 31 30 37
22 34 39 38 26 6 31
23 5 28 40 32 30 37
24 3 4 36 26 6 31
25 28 17 7 31 30 37


26 40 33 7 30 27 37
27 30 16 10 26 23 17
28 40 23 7 30 37 27
29 30 10 1 15 23 6
30 29 27 16 15 23 17


31 32 39 14 30 10 37
32 37 4 3 23 6 15
33 40 26 7 31 27 32
34 22 21 2 27 37 31
35 18 23 40 27 4 32



These results yield some interesting points. The computer, as expected, is very literal in it's analysis of the images. For example, look at the most similar matches for image 13. At first, it was a surprise to see it match with 10 and 1. As a human, I look at 10 and 1 and categorize the images as green. The computer, however, is not swayed by what we might see as the primary colors of the image and recognizes that both of these images have quite a bit of shades that are closer to white. Looking at image 10, the way the light is hitting the stem of the broccoli gives it some very light shades of green that will be close to white. If we think of the histograms computed earlier, it is quite feasible to think that it had pixels in the same bin that most of the white pixels of image 13 fell in (hence, the match).

Image 31 doesn't seem to match well with many of the images. Looking at the images most similar to it, it appears that it tried to match with shades that would be fairly close in RBG values. Considering the system, this makes sense since the bins will allow for the overlap. And even if the matches do not appear to close, they may have simply been the closest that this batch of images had to offer. This is reflected in the 40 x 40 color pairwise comparison matrix. The average similarity between two images was 0.17. 31 did match with 32 with a similarity of 0.20, but it quickly feel off with its second match against 39 with a similarity of 0.04. Because of this lack of similarity with most images, it's no surprise that it showed up quite regularly in one of the top three most dissimilar images for the other samples. In fact, it showed up one of the dissimilar positions for 20 of the other images.

Another fact that stood out to me is that the data set contains a good number of green and red images, and these seemed to be classified as counterparts for one another. In other words, they were obviously similar to others with the same color, but greens tended to be dissimilar to reds and reds were dissimilar to greens. This was particularly true with images that were primarily just one of these shades. For example, images like 6 and 37 were pretty extreme shades of red and green, respectively. There wasn't much noise in their images that would allow them to match up well with images that weren't close to their shade. An image like 5 was more of a wildcard, though, given its mixture of red and green. While predominately red and matched with more red images, it didn't appear once in the "dissimilar" columns because it was more likely to have overlap with other images.

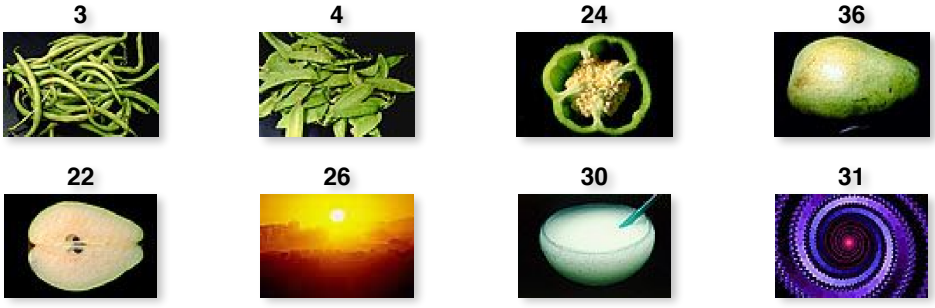
Overall, analyzing these outcomes was a lesson in thinking like a computer. You really have to image how the system is placing an images pixels in different bins and how different image histograms could match up (sometimes for obvious reasons, sometimes because there simply isn't a better option).

Finding the Four Most Similar/Dissimilar - **getSimilarityGroup(cmps, opts)**

The last part of this step requires finding the four images that are the most similar to each and the four images that are the most dissimilar. The function accomplishes this using the 40 x 40 comparison matrix calculated above so there is no need to recompute similarity between images.. It can be used for calculating the most similar and least similar simply by passing in one of the enumerated options I defined (Opts.Similar or Opts.Dissimilar).

The function determines the best group of four images by iterating through all combinations of four images. These combinations are given a score that is equal to the sum of all possible pairwise comparisons of the four images in the group. If we are looking for the most similar group, the algorithm remembers group with the highest score and returns it at termination. If we are looking for the most dissimilar, the algorithm remembers the group with the lowest score and returns it instead. The output is seen below with image numbers above the image.

*The top row represents the most **similar** group of four images, the bottom row represents the most **dissimilar** four images.*



The images in each group seem quite reasonable. Looking at the “most similar” group, a human might just see a bunch of green images (that just happen to be fruits/vegatables) but the system surely saw much more than this. Specifically, for each image, it likely saw the darker shades of green in one bin while catching the lighter green / yellowish green pixels in another. We can notice that each of these images shares these variations, so it’s no surprise that they would have had high similarities based on our **l1Compare()** function in ever possible pairwise comparison.

Looking at the “most dissimilar” group, we can see that the images vary quite a bit in color. When forming the color histograms, it’s easy to image how these four images varied quite a bit in how the majority of their pixels were classified. For example, the bluish/purple pixels that dominate 31 likely shared little to no overlap with the redish orange pixels that dominate 26 or the yellowish green pixels in 22. Without this overlap, the similarities remained low, which led the system to choose this combination.

If we look to the three most similar and three most dissimilar output in the previous step, we can see a lot of agreement here. The images in the most similar group showed up in each other’s “most similar” columns, while the images in the most dissimilar group showed up in each other’s “most dissimilar” columns.

Although these results appear perfectly reasonable, I tried determining these most similar/dissimilar groups using another score. Consider looking for a group of most similar. I was worried that the algorithm might output a group where three of the images had high similar scores, but the fourth closest images to these three was one that was noticeably dissimilar. This could occur if the three images were so close to each other that the sum of their score was enough to beat out another fairly similar group of four images. To the computer, this makes sense based on the scoring mechanism it was provided. A human, however, might be perplexed to see three very similar images with one that isn’t so similar. In fact, the human may even think a set of four images that aren’t as similar but that doesn’t have a huge outlier makes more sense.

To accommodate this, I tried an algorithm that assigned a group a similarity score equal to the minimum pairwise similarity score of all the images it contained. And when looking for the most dissimilar group, it assigned scores equal to the highest pairwise comparison of the images in its group. However, this method still ended up outputting the same results, so I decided to stick to the original. I think it would be a little more crucial if fewer, more diverse images were stored in the database. I also think it would play greater importance if we tried to form larger groups of “most similar/dissimilar” images.

Step 2 - Gross Texture Matching

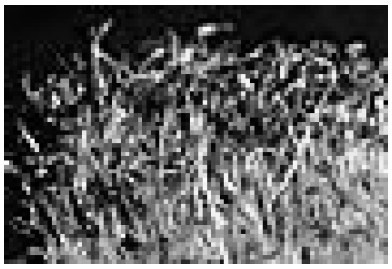
We continue how by explaining how the system compares images to each other based on texture. This text refers to the **%% Step 2** section of the code in **HW2.m**. It will produce output identical to that of Step 1, but will use texture as its metric instead of color.

Texture will be based on the Laplacian of the images, which we can calculate form the gray scales of the RGBs. Let's walk through the process of converting these images and then doing the analysis.

Converting RBGs to Gray Scales - **getGrayScale(img)**

The first thing we must do is convert the images into gray scales so that we can calculate the Laplacians. I chose to follow the assignments recommendation and calculate gray scales for an RGB image by taking the average of the red, green, and blue values [(r + g + b) / 3] for each pixel.

Gray Scale of Image 1



Converting the Gray Scales into Laplacians - **getLaplacian(gray)**

With the gray scales in hand, we can then calculate the Laplacian for each image as described in the assignment details. The function also handles pixels in a corner and on the side of an image different. For corner pixels, the value of the pixel in question is multiplied by three and then the three pixels around it are subtracted from it. For side pixels, the value of the pixel in question is multiplied by five and then its five surrounding pixels are subtracted from it. All other pixels used the typical procedure provided.

Another important note is that all Laplacian values were stored as positive numbers (the absolute value is calculated before storage). An important exception to this is the background pixels, which are stored as -1 to distinguish themselves from the other values.

Using my observations from Step One, I decided to treat pixel values with a gray scale value less than 37 as the background. Although this could treat certain pixel values as background that were treated as the foreground in the color analysis (such as 60, 0, 0), deviating from this number resulted in slightly less favorable scores against the user rankings (seen below). These scores are combined most similar and dissimilar, so they are out of 240. The selection of

texture bins hasn't been covered yet, but just know that they were held constant at 350 for now so the results are comparable.

Gray Scale Threshold	25	37	45
User 1	18	15	17
User 2	38	42	42
User 3	41	43	41
User 4	51	51	44
Average	**37**	**37.75**	**36**

Lastly, this function also returns the maximum value in the Laplacian image. The maximum value of all of the images is used as the upper bounds for the histogram calculated in the next step.

Creating Texture Histograms - **getNormalizedTextureHistogram(laplacian, bins, max)**

With the Laplacians at hand, we can now create histograms for each image based on their corresponding Laplacian images. Higher texture images will have Laplacian images with more high values while lower texture images will have Laplacian images with more low values. We also have to account for images with more background than others as this background, if not handled appropriately, could make an image appear less textured than it is.

This function initializes a one dimensional histogram with 350 bins. The size of the bins are determined by taking the maximum value of any of the Laplacian values and dividing it by 350. In this case, the maximum vale is 1,520 so each bin has a range of about 4.3. A count for the total number of non-background pixels is also initialized to 0.

Note: *The number of bins was determined based on how well the system scored against the user data. These scores are the combined total for similar and dissimilar matches (out of 240). Details of the scoring system are found in Step 0.5. Overall scores for color matching at found in Step 4.*

# Bins	20	50	100	200	350	500
User 1	9	11	12	13	15	15
User 2	38	39	37	34	42	39
User 3	32	38	33	34	43	37
User 4	46	45	43	46	51	48
Average	**31.25**	**33.25**	**31.25**	**31.75**	**37.75**	**34.75**

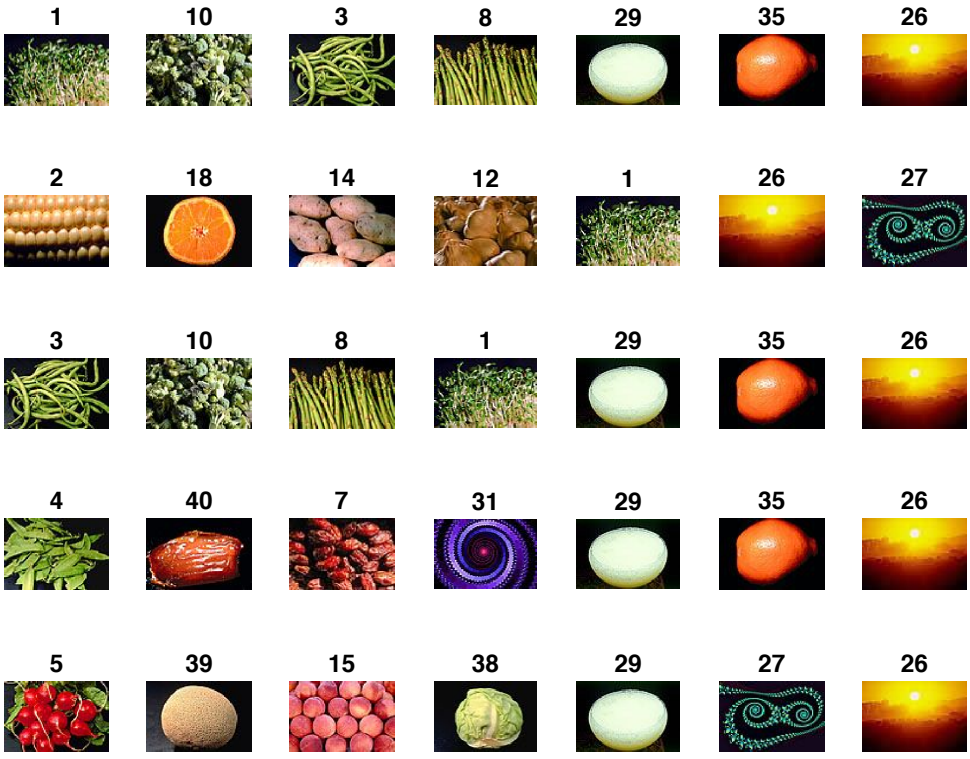
The function then goes through each pixel in the Laplacian image. If the pixel value is not equal to -1, then it is **not** the background. In this case, the total number of non-background pixels is incremented and the value is placed into the appropriate bin by incrementing the bin's count. The appropriate bin is found by taking the ceiling of the pixel value divided by the bin range (with the corner case of a value of 0, which is assigned to the first bin).

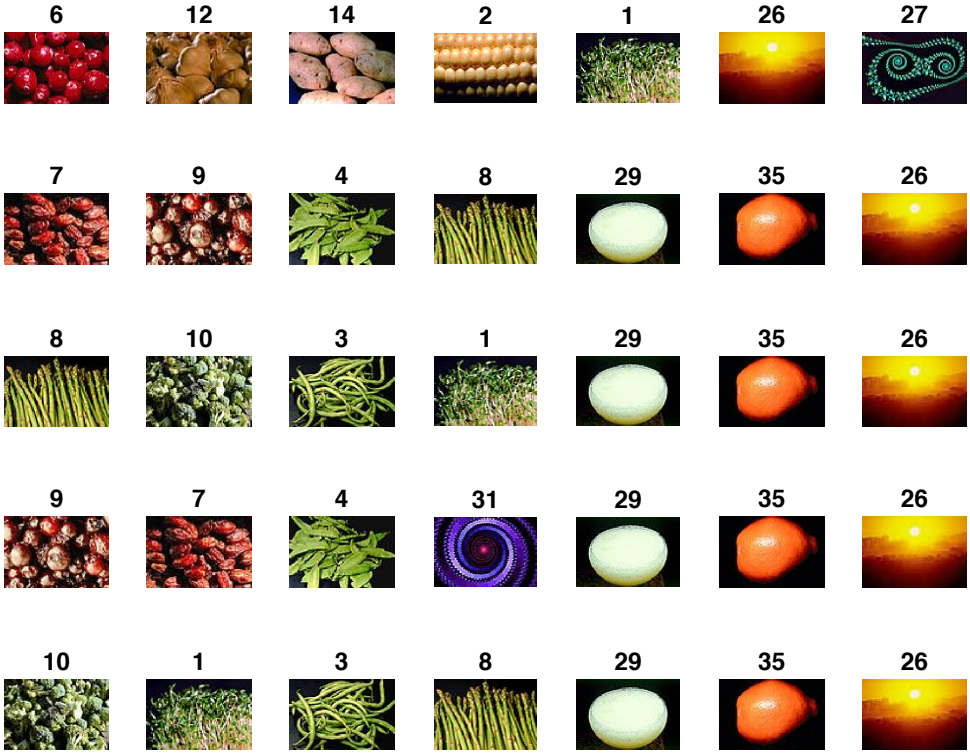
Once all of the pixels have been placed into the histogram, it is normalized by dividing each value in the histogram by the total number of non-background pixels. As in the color histograms, this makes the values in the range [0, 1] and represents the percentage of pixels from the Laplacian image that fell in that bin.

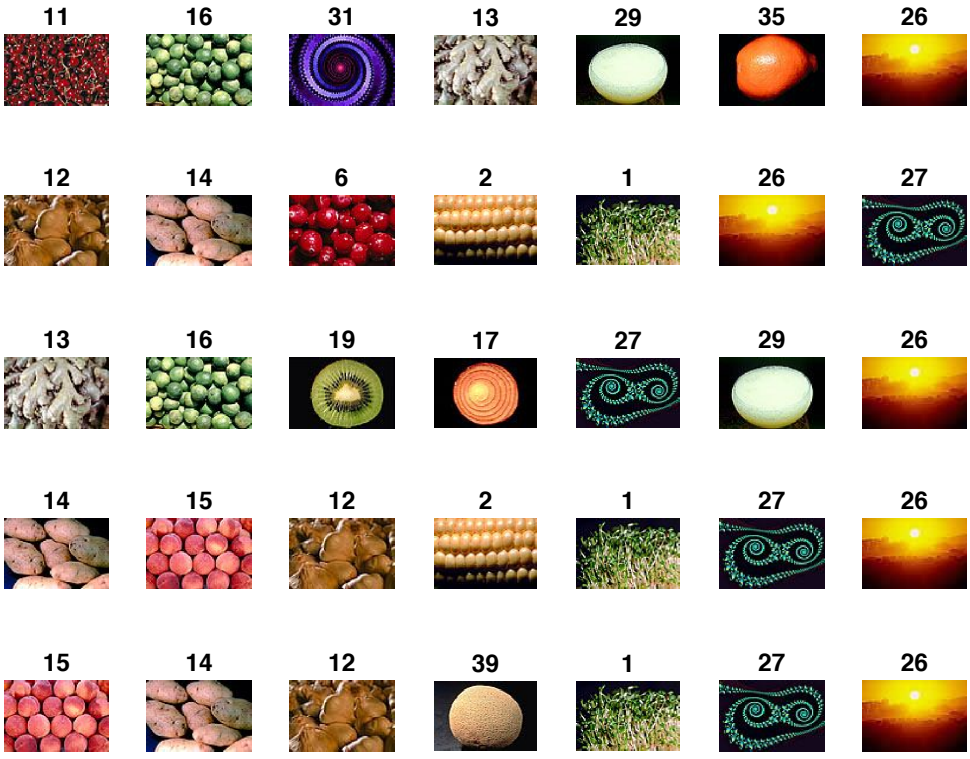
Getting the Similarity Results - **l1Compare(vec1, vec2)** and **getSimilarityResults(cmps)**

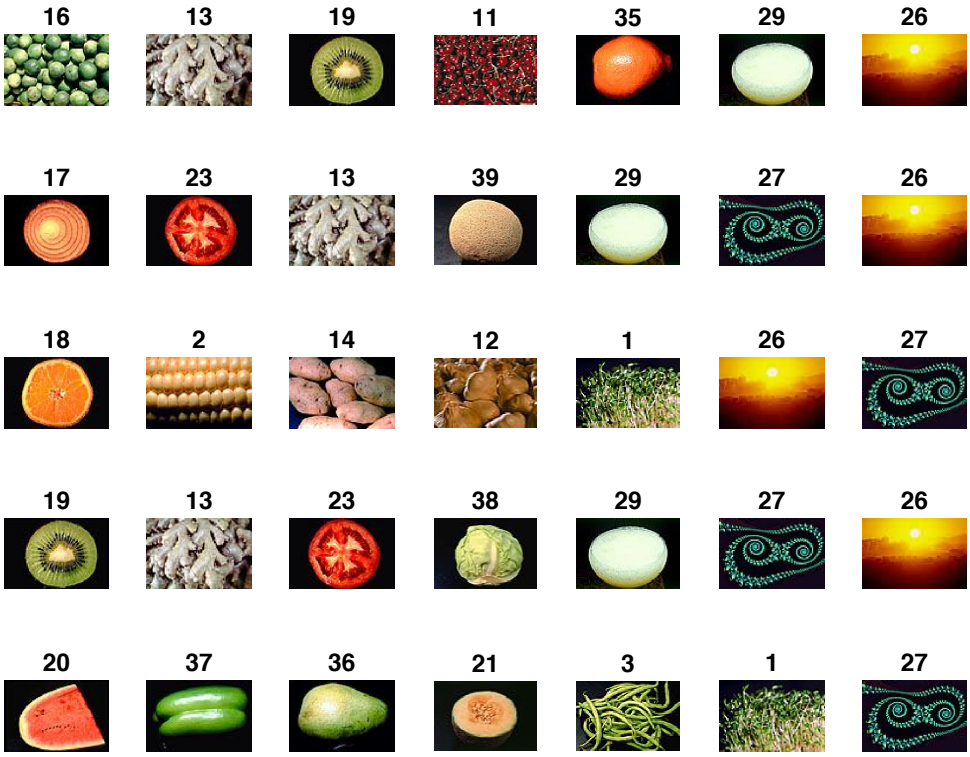
The exact functions used to create the 40 x 40 color comparison matrix and compute the 40 x 7 matrix where each row consist of the image at hand, the three most similar, and three most dissimilar can be reused for texture. Given that, I'll omit repeating the discussion and direct the reader to the section on **Step 1** for more information.

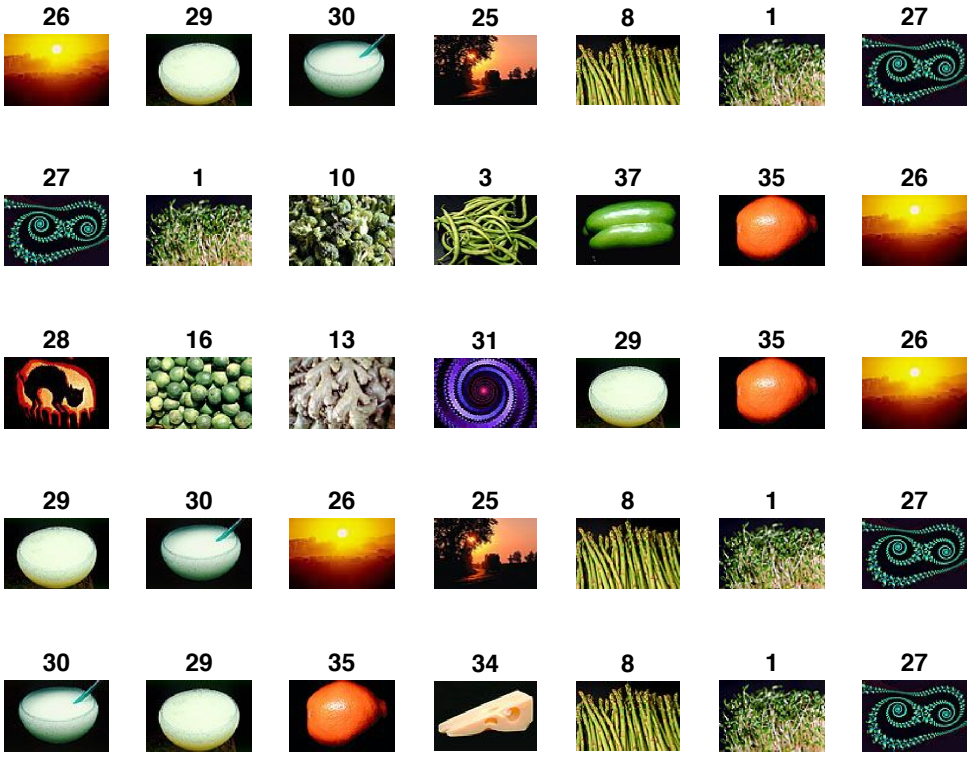
The system will create the 40 x 40 texture comparison matrix from the texture histograms found above. This matrix is then used to produce the the output for the similarity comparisons on texture:

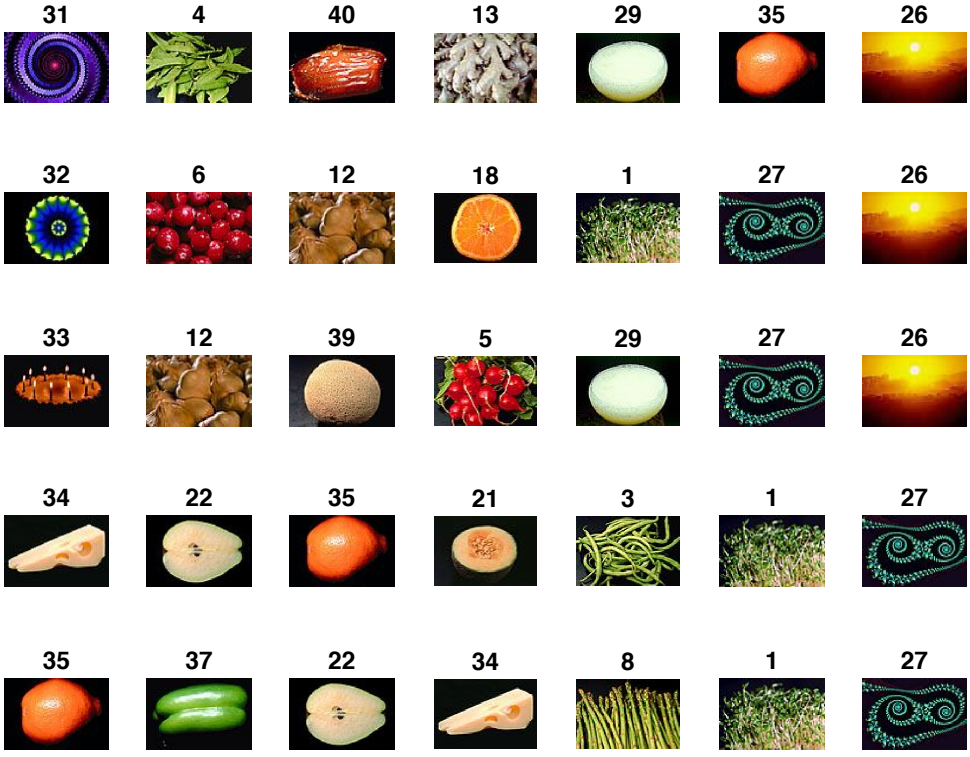


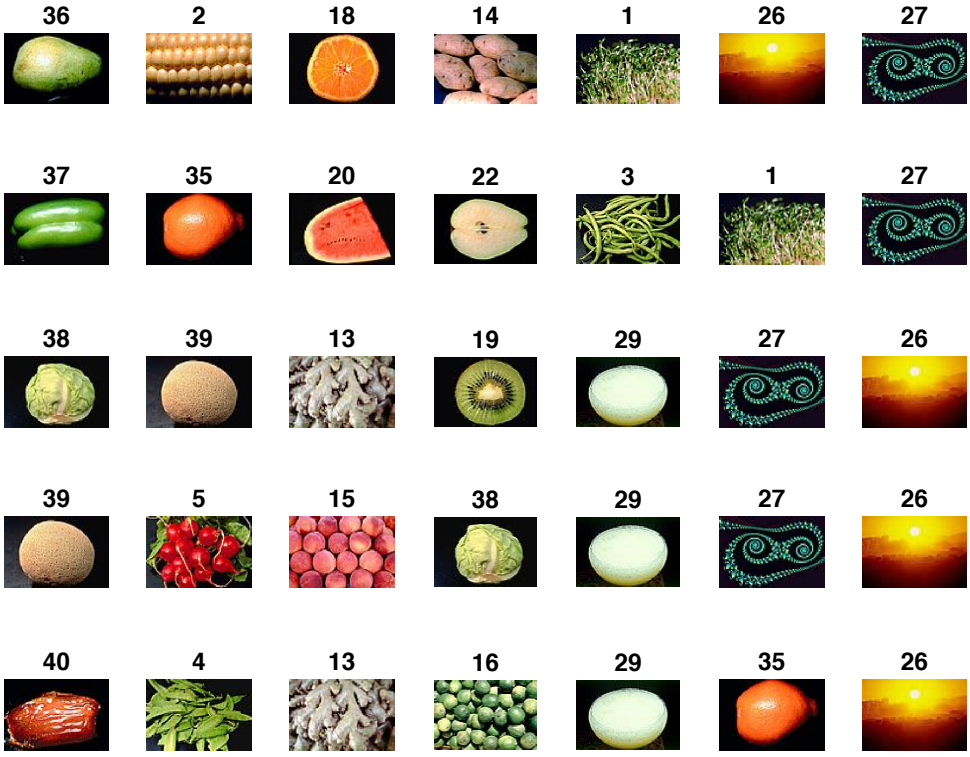

6 12 14 2 1 26 27
7 9 4 8 29 35 26
8 10 3 1 29 35 26
9 7 4 31 29 35 26
10 1 3 8 29 35 26


11 16 31 13 29 35 26
12 14 6 2 1 26 27
13 16 19 17 27 29 26
14 15 12 2 1 27 26
15 14 12 39 1 27 26


16 13 19 11 35 29 26
17 23 13 39 29 27 26
18 2 14 12 1 26 27
19 13 23 38 29 27 26
20 37 36 21 3 1 27
21 12 6 20 1 26 27
22 35 34 37 8 1 27
23 17 13 19 29 27 26
24 16 4 31 35 29 26
25 29 30 5 36 37 27


26 29 30 25 8 1 27
27 1 10 3 37 35 26
28 16 13 31 29 35 26
29 30 26 25 8 1 27
30 29 35 34 8 1 27


31 4 40 13 29 35 26
32 6 12 18 1 27 26
33 12 39 5 29 27 26
34 22 35 21 3 1 27
35 37 22 34 8 1 27



These results take a little more thought that the color comparisons, but after some consideration, it's easy to see that the system is making reasonable matches given the instructions it was given. In many cases, it's amazingly accurate and establishing similarities that a human would likely make.

Consider the first row for image 1. Image 1 itself ranks higher on the texture list because of the gaps in its leaves and its bushy form. This two traits result in lots of internal edges that lead to high Laplacian values. This can be seen as well with its most similar matches: 10, 3 and 8. All of these pictures have quite a bit of activity inside of them that, again, creates noticeable edges and higher texture values. This is in contrast to the most dissimilar matches, 29, 35, and 26. Each of these have very smooth gradients and solid colors within them. Because they have relatively few internal edges, the texture ranking is low, as one would expect.

However, there are some where it takes a little more thought to understand the output. Take the second row for image 2. Relative to the rest of the images, image 1 has a texture ranking that comes in around the middle of the pack. It's matches with 14 and 12 are quite understandable as these two have quite a few internal edges, but not as much as 1, 10, and 3. Matching closest to 18 takes a little more thought but we have to remember that the system is looking very closely at the image (at each pixel - closer than any human would look) so it's picking up on all of the little lines and spot the inside of an orange might have. The "middle of the pack"

grouping is supported by the dissimilar matches, 1, 26, and 27. 1 and 27 are highly textured images while 26, as discussed above, has the smooth gradient that makes it low in texture. Given that picture 2 is dissimilar to images that fall at both extremes, I have to think that puts it somewhere in the middle.

Overall, analyzing the texture of the images seems to work well and will give us an additional feature when it comes time to form sensible clusters.

Finding the Four Most Similar/Dissimilar - **getSimilarityGroup(cmps, opts)**

Once again, we get to take advantage of the same function developed for **Step 1** by feeding in the 40 x 40 texture comparison matrix into this function. In the case of texture, the function returns the following two groupings:

*Top row represents most **similar** in texture; bottom row represents most **dissimilar** in texture.*

The first row just so happens to be the same images discussed above for image 2. If we look at the results for each of these images, we'll see that they all contain each other, with just a few exceptions, in their most similar columns. Although we can make sense of this based on the formula we had the system use, a human use might find it hard to see that these are all similar in texture as they are quite familiar with these objects in real life. 12 and 14 might be reasonable, and perhaps 2 isn't too far from those either, but the inside of an orange (18) might be a bit of a stretch.

The bottom row seems quite reasonable. 27 would be considered the most textured image with lots of internal edges. Of these four, 8 would be the 2nd most textured as it has quite a few internal edges itself, but not as many as 27. Finally, 26 and 37 are relatively smooth images with colors that change in a smooth gradient. Group together, it's reasonable from the Laplacian and human perspective that these images would be quite dissimilar in texture.

For more details on how the grouping works, please see the corresponding section in **Step 1** as the function remains the same.

Step 3 - Combine the Similarities and Cluster

We continue our analysis by combining the color and texture similarities to form clusters of the images. Clusters will be formed by putting certain emphasis on color and texture. For example, we may put 80% emphasis on color and 20% emphasis on texture. These proportions are explained by the variable *r*, which we'll adjust to weight the characteristics differently. Specifically, color will be weight by (1-r)% and textured will be weighted as r% where r is in the range [0, 1].

We'll do clustering for both complete link and single link as definitions for "nearness."

This text refers to the **%% Step 3** section of the code in **HW2.m**.

Initializing Variables

The algorithm begins by specifying values for *r* and the number of clusters (7). It also calculates S, which is the linear sum of r * [texture comparison matrix] + (1-r) * [color comparison matrix]. This is a 40 x 40 matrix with values between [0, 1] where values closer to 1 imply greater similarity between the images at row i and column j.

It then calculates D by taking every element in S and subtract it from 1. This resulting matrix is also 40 x 40 where the values are meant to represent distances between points representing the images of row i and column j. That is, lower numbers are supposed to mean the image points are closer together. This is useful for clustering as we want to group images by "how far apart" they are.

Clustering

I encountered two ways to cluster the images represented by the pairwise distance matrix D. The first was by using MATLABs functionality as described in this section. The second is my own clustering function that explicitly follows the instructors algorithm.

Although MATLABs functionality offered support for complete link and single link, the results were ever so slightly better from my own implementation. Although my first thought was to think that something was wrong with my code, my implementation's output was quite reasonable and I couldn't find any bugs in it. Instead, I started to wonder if MATLABs functionality is doing a bit more work than it should be. With that in mind, I decided to present both as I don't want to mistakenly assume MATLABs implementation does what the instructions asked. I'll focus on my own implementation when doing the analysis.

Clustering - MATLAB functionality - **HW2.m**

Clustering is quite easy using MATLABs functionality. As shown in the code, you use the **linkage()** function to create a matrix that encodes a tree of hierarchical clusters of rows of the pairwise distance matrix D calculated above. **cluster()** is then used to take the output from **linkage()** and the maximum number of clusters (7) to assign the images (1 - 40) in a cluster.

Clustering - My Own Implementation - **clusterSimilarities(D, num_clust, option)**

My implementation follows the assignments instructions explicitly. You can pass in either Opts.Complete or Opts.Single to dictate how distance is calculated between two existing clusters.

The function begins by assigning each image to its own cluster. Then, while the total number of clusters is greater than 7, the algorithm finds the two clusters closest together by iterating through all pairwise combinations of them. Again, the distance between clusters is defined by the option parameter and the local function **getDist()** calculates it appropriately by doing all pairwise comparisons between the images in the two clusters being analyzed. The two clusters that are closest together are then combined by adding all of the images from one cluster to the other cluster (at a high-level, this is something of a naive implementation of a Union-Find data structure). These iterations that combine clusters continue until 7 clusters remain.

Memoization is used to avoid recomputing distance between clusters when this distance between the clusters was determined in a previous iteration. However, when two clusters are combined, this distance is set to -1 for all comparisons involve these clusters so that the algorithm knows the distance needs to be recomputed (since the clusters changed).

At conclusion, all 40 images will be assigned to one of seven clusters.

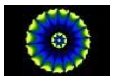
A Sanity Check - Complete Link

To get a feeling for the performance of the two methods, let's consider r = 0 (all weight on color) and see the two algorithms cluster the images using the complete link method. All emphasis is put on color not only because it is easier to see if the algorithms are behaving appropriately (compared to texture clustering), but also because it's quite reasonable to think that users would group the images by color. The output from these clusters are on the following pages.

As we can see, the two methods both seem to do well in clustering images on their color from a visual perspective. The complete link method does a fine job of forming clusters where no two elements within a group are too different from each other. Although we'll see that the clusters formed by the MATALB functions will perform slightly better when scored against the user clusters, the difference was not so drastic that it made me want to abandon my own implementation. More importantly, I completely understand what my own implementation is doing, while MATLABs algorithms might be up to something outside of the scope that the instructions described.

*Complete Link Clustering with **MATLAB Functions** - r = 0.0*

Cluster 1

27 
32 
37 

Cluster 2

13 
29 
30 

Cluster 3

31 

Cluster 4

17 
18 
25 
35 

Cluster 5

2 
12 
14 
20 
21 
22 
34 

39 

Cluster 6

5 
6 
7 
9 
11 
15 
23 

26 
28 
33 
40 

Cluster 7

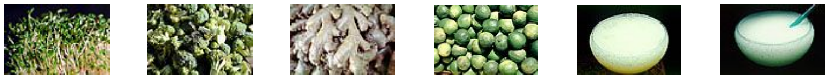
1 
3 
4 
8 
10 
16 
19 

24 
36 
38 

*Complete Link Clustering with **My Function** - r = 0.0*

Cluster 1

1 10 13 16 29 30



Cluster 2

2 17 18 20 21 25 35

Cluster 3

3 4 8 19 24 36 38

Cluster 4

5 6 7 9 11 15 23

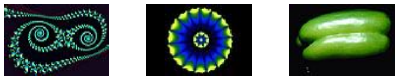
26 28 33 40

Cluster 5

12 14 22 34 39

Cluster 6

27 32 37



Cluster 7

31

A Sanity Check - Single Link

Next, we'll show the single link output for the two algorithms. It's reasonable to think that the single link method won't create the most distinct clusters given the way it defines nearness. Instead, we could expect it create clusters that might contain pairs of elements that are quite a bit different from each other, but can be "linked" to each other through a path of more similar items also within that cluster. As the clusters form, you can image a spanning tree linking up nearby nodes.

The MATLAB functions seemed to do a better job of grouping the images that were noticeably different from each other into different clusters. For example, cluster 1 and 2 outputted by the MATLAB methods were essentially put into one cluster (number 2) in the output from my implementation.

*Single Link Clustering with **MATLAB Functions** - r = 0.0*

Cluster 1

5 6 7 9 11 15 17

18 23 25 26 28 33 35

40

Cluster 2

2 12 13 14 21 22 34

39

Cluster 3

20

Cluster 4

27 

29 

30 

Cluster 5

1 

3 

4 

8 

10 

16 

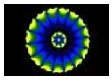
19 

24 

36 

38 

Cluster 6

32 

37 

Cluster 7

31 

*Single Link Clustering with **My Implementation** - r = 0.0*

Cluster 1

1 
3 
4 
8 
10 
13 
16 

19 
24 
36 
38 

Cluster 2

2 
5 
6 
7 
9 
11 
12 

14 
15 
17 
18 
20 
21 
22 

23 
25 
26 
28 
33 
34 
35 

39 
40 

Cluster 3

27 

Cluster 4

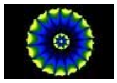
29 
30 

Cluster 5

31

Cluster 6

32



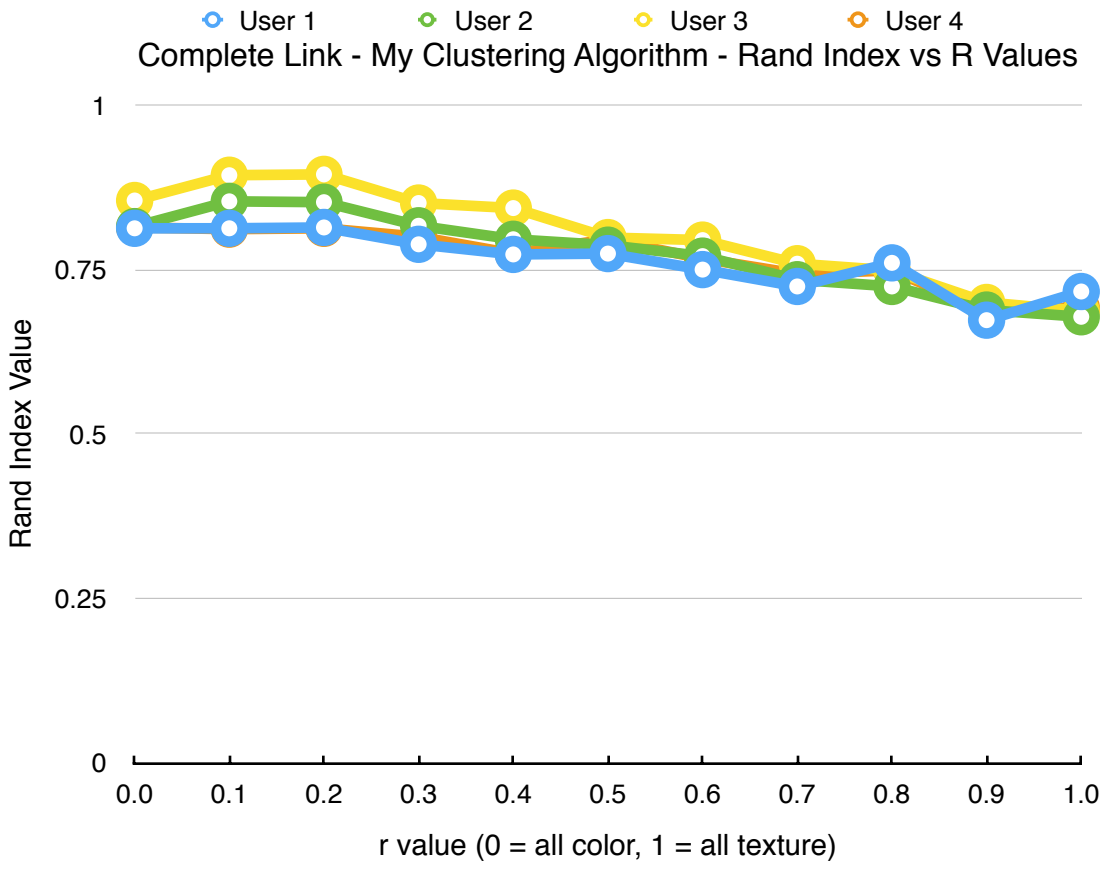
Cluster 7

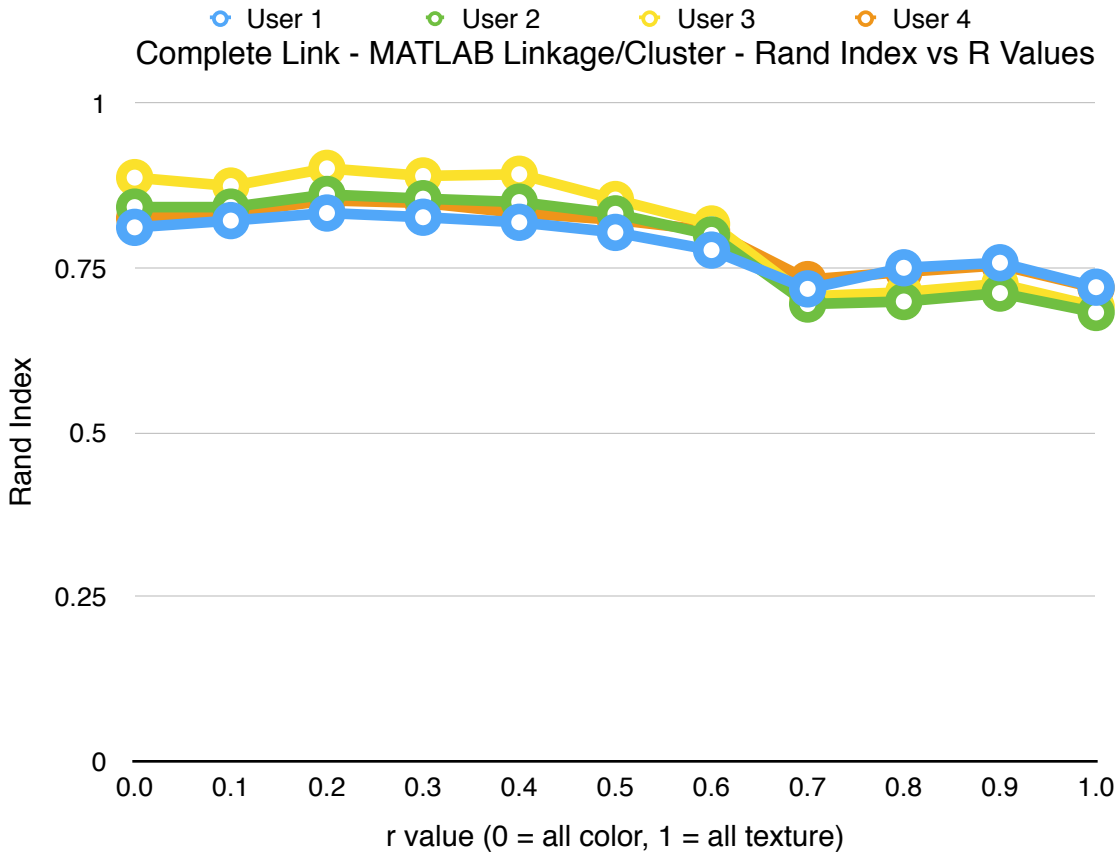
37

Complete Link and Varying Values of R

Although r = 0.0 was a good starting point, we want to try to find a value for *r* that forms clusters that score well against the user clusters. In search of such a value, I ran the clustering algorithms (both my own and MATLABs) using values from r = 0.0 to 1.0. The resulting clusters for each output were compared to all four users and a Rand index was calculated to determine similarity. These graphs are shown below (my clustering algorithm on top, MATLABs on bottom).

In both cases, the Rand index appeared greatest when r was closer to 0.0, or more weight was put on the color similarities. Based on the two graphs, it seems reasonable to say that r = 0.2 was the optimal value to get the system to output clusters more similar to the users (based on the Rand index).

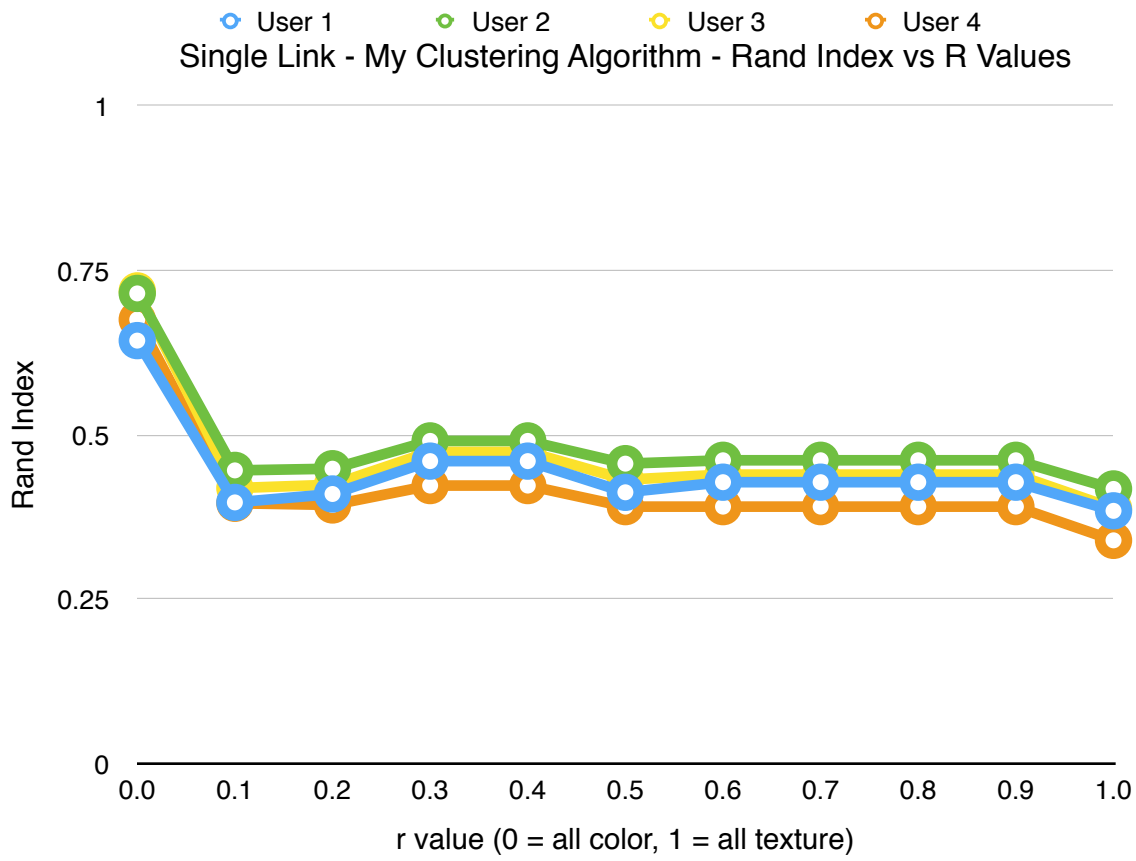


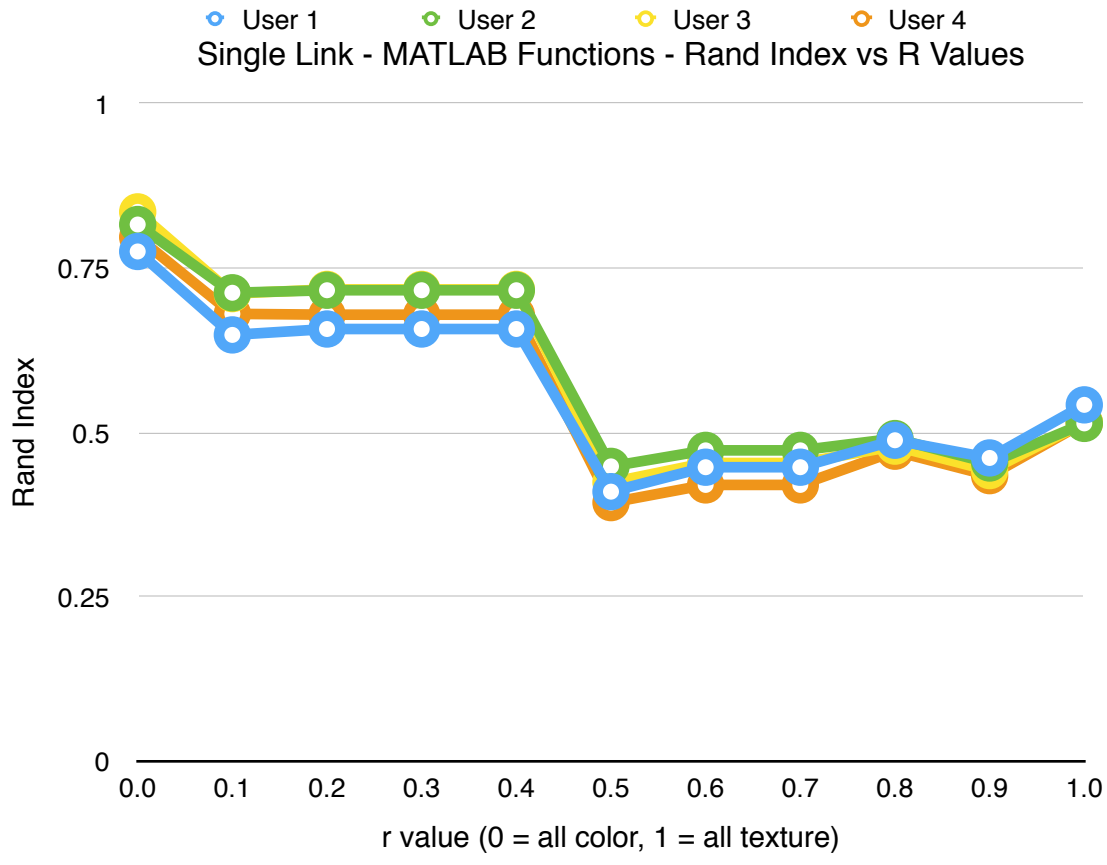


Single Link and Varying Values of R

The similar analysis as above was also performed on single link clustering methods. The two graphs for my implementation (top) and MATLABs functions (bottom) are shown next.

What's most noticeable is that the clusters with lower r values, specifically, r = 0.0, scored best against the user input. The Rand index values deteriorated quickly as r grew to include more texture.


User 1
User 2
User 3
User 4
Single Link - My Clustering Algorithm - Rand Index vs R Values
Rand Index
1
0.75
0.5
0.25
0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
r value (0 = all color, 1 = all texture)


User 1
User 2
User 3
User 4
Single Link - MATLAB Functions - Rand Index vs R Values
Rand Index
1
0.75
0.5
0.25
0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
r value (0 = all color, 1 = all texture)

Conclusions on Clustering

A couple things jump out based on the graphs above. First, performance tends to improve when more weight is put on color of texture. Second, according to the Rand index metric, clusters produced using complete link to define nearness tend to be more similar to the user clusters than the single link method. Furthermore, given the choice of complete link to define nearness, choosing an r value of 0.2 seems to produce the best output (again, relative to the user clusters using the Rand index).

Given these results, the clusters using my clustering implementation, complete link method, and r = 0.2 are shown below.

Cluster 1

1 3 4 8 10 16 19

24 32 36 37 38

Cluster 2

2 17 18 21 25 35

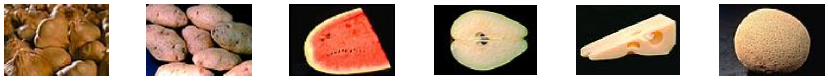
Cluster 3

5 6 7 9 11 15 23

26 28 33 40

Cluster 4

12 14 20 22 34 39



Cluster 5

13

29

30

Cluster 6

27

Cluster 7

31

Step 4 (Step 4b) - Discussion of Results

Similarity Matching

Given all of the implementation details set based on the above discussion, the system scored as follows on color and texture similarity matching.

	Color - Similar	Color - Dissimilar	Texture - Similar	Texture - Dissmilar
User 1	36	4	14	1
User 2	34	34	20	22
User 3	57	45	24	19
User 4	57	32	32	19
Average	46	28.75	22.5	15.25
Percentage	38.33%	23.96%	18.75%	12.71%
Combined	74.75 / 240 = 31.14%		37.75 / 240 = 15.72%	

The system seems much better at matching the users on color than on texture. This may be true because of the strict definition we gave the system for measuring texture. While the system essentially looks for sharp changes in color as these represent edges, the user was probably familiar with many of the objects in the images provided. For example, while the system might view corn and an the inside of an orange as having the same texture because of the number of lines/edges they contain internally, a human can tell the difference between the juice fruit at the smooth bumps of the kernels. Color, however, is much less ambiguous. We programmed the system to essentially see and group color the way many humans would.

Another interesting note is that the system scored much better in matching the user on similarities rather than dissimilarities. This perhaps isn't too surprising. Most people can converge pretty quickly on saying that different shades of green are similar. However, given a shade of yellow, they may not be so sure whether that yellow is more different from a true blue or dissimilar to a true red. One persons opinion might even change by the day. The system, however, has very clear definitions for these decisions and is much more consistent.

Finally, another major factor is that the human likely did not carefully compare each image to every other image in making their decision. Say the images were presented to the user in 8 rows of 5 images. If the user was first judging the image in the first row and first column, the user may simply look at the nearby images when declaring the one that is the most similar or dissimilar. The system, however, does not have this spatial bias. In a sense, it compares each image to every other image with equal importance before making its decision.

Clustering

Most of the details and results for cluster were discussed in **Step 3**, but we can conclude here by saying that the complete link method with an r value of 0.2 proved to cause the system to

output the most similar clusters to the users (according to the Rand index). The Rand index values comparing the system output to the user output are given below. They are based on the complete link method and an r value of 0.2 for each of the clustering methods.

	MATLAB Functions	My Implementation
User 1	0.83	0.81
User 2	0.86	0.85
User 3	0.90	0.89
User 4	0.85	0.81
Average	0.86	0.84

I think that some of the biggest discrepancies was probably cause by factors not included in color or our definition of texture (again, simply having sharp lines within an object). For example, it would be perfectly reasonable to put all vegetables in one cluster and fruits in another, even if the colors were not similar. Shapes may have also played a role as one of my users specifically said “I made this cluster because all of the objects were round.”

Ultimately, however, color was a noticeably dominate factor in the user clusters, which gave our system a good shot and outputting something similar.

References

In order to sort a matrix by a particular column, I reference this open Matlab Central support thread: https://www.mathworks.com/matlabcentral/newsreader/view_thread/248731

In learning about different MATLAB data structures, specifically the cell array, I reference this document out of Carnegie Melon: http://matlab.cheme.cmu.edu/2011/09/26/some-basic-data-structures-in-matlab/

In learning about the Rand Index, I reference this Wikipedia page: http://en.wikipedia.org/wiki/Rand_index

Everything else involved a heavy reliance on the MATLAB documentation located here: http://www.mathworks.com/help/matlab/

Appendix

*******HW2.m*******

```
clc; clear; close all;

%% Step 0

% debug
print_color_results = false;
print_texture_results = false;
print_user_input = false;

% Get the image filenames
imgPath      = 'ppm/';
imgType      = '*.ppm'; % change based on image type
imgFiles     = dir([imgPath imgType]);
rgbs         = cell(length(imgFiles), 1);
N            = length(imgFiles);

% Load images
for i=1:N
    filename = [imgPath imgFiles(i).name];
    rgbs{i} = imread(filename);
end

%% Step 1

% Number of segments to divide 255 channel into; bins = segments^3
num_segments = 7;

% Create array of normalized histogram vectors for each image
color_hists = zeros(N, num_segments^3);
for i=1:N
    hist3D = getNormalizedColorHistogram(rgbs{i}, num_segments);
    color_hists(i,:) = reshape(hist3D, [1 num_segments^3]);
end

% Perform comparisons between images
color_cmps = zeros(N,N);
for i=1:N
    for j=(i):N
        comp = l1Compare(color_hists(i,:), color_hists(j,:));
        color_cmps(i,j) = comp;
        color_cmps(j,i) = comp;
    end
end

% Gets results with image names as specified by assignment (40 x 7)
color_match_results = getSimilarityResults(color_cmps);

% Get the four most similar and dissimilar based on color
color_most_similar = getSimilarityGroup(color_cmps, Opts.Similar);
```

```
color_most_dissimilar = getSimilarityGroup(color_cmps, Opts.Dissimilar);
overall_color_matches = [color_most_similar; color_most_dissimilar];

% Print results of comparisons
if print_color_results
    printResultsWithImages(color_match_results, rgbs);
    printResultsWithImages(overall_color_matches, rgbs);
end


%% Step 2

% Get gray-scale images
grays = cell(N, 1);
for i=1:N
    grays{i} = getGrayScale(rgbs{i});
end

% Get Laplacian images
laplacians = cell(N, 1);
max_l = 0;
for i=1:N
    [laplacians{i}, max_val] = getLaplacian(grays{i});
    max_l = max(max_l, max_val);
end

% Turn Laplacians into histograms
bins = 350;
text_hists = zeros(N, bins);
for i=1:N
    text_hists(i,:) = getNormalizedTextureHistogram(laplacians{i}, ...
        bins, max_l);
end

% Perform texture comparisons between images
texture_cmps = zeros(N,N);
for i=1:N
    for j=(i):N
        comp = l1Compare(text_hists(i,:), text_hists(j,:));
        texture_cmps(i,j) = comp;
        texture_cmps(j,i) = comp;
    end
end

% Gets results with image names as specified by assignment (40 x 7)
texture_match_results = getSimilarityResults(texture_cmps);
% Get the four most similar and dissimilar based on texture
text_most_similar = getSimilarityGroup(texture_cmps, Opts.Similar);
text_most_dissimilar = getSimilarityGroup(texture_cmps, Opts.Dissimilar);
overall_text_matches = [text_most_similar; text_most_dissimilar];

% Print results of comparisons
if print_texture_results
    printResultsWithImages(texture_match_results, rgbs);
```

```
    printResultsWithImages(overall_text_matches, rgbs);
end


%% Step 3
% Cluster the images using Complete and Single Links

% Determine pairwise distances
r = 0.2;
S = r * texture_cmps + (1.0 - r) * color_cmps;
D = 1 - S;
n_clusters = 7;

% OPTION 1: % Use MATLAB's Agglomerative Hierarchical Cluster Tree funcs
% Group the data using linkage
% Z = linkage(D,'complete');
% sys_c = cluster(Z,'maxclust',n_clusters);
% for i=1:n_clusters
%     mat = vec2mat(find(sys_c == i), 7);
%     printResultsWithImages(mat, rgbs);
% end

% OPTION 2: Self-made clustering algorithm
sys_c = clusterSimilarities(D, n_clusters, Opts.Complete);
for i=1:length(sys_c)
    mat = vec2mat(find(sys_c == i),7);
    printResultsWithImages(mat, rgbs);
end

%% Step 4

n_users = 4;

% Initialize scores array for three users [usr colors textures clusters]
scores = [1 0 0 0 0 0; 2 0 0 0 0 0; 3 0 0 0 0 0; 4 0 0 0 0 0];

% Load user surveys for first two steps
surveys = cell(n_users,1);
surveys{1} = csvread('data/SurveyOne.csv');
surveys{2} = csvread('data/SurveyTwo.csv');
surveys{3} = csvread('data/SurveyThree.csv');
surveys{4} = csvread('data/SurveyFour.csv');

% Print images
if print_user_input
    for i=1:n_users
        printResultsWithImages(surveys{i}, rgbs);
    end
end

% Get survey scores
for i=1:n_users
    submitted = surveys{i};
    scores(i,2:3) = getScore(color_match_results(:,2:7), submitted(:,2:3));
```

```
    scores(i,4:5) = getScore(texture_match_results(:,2:7), submitted(:,4:5));
end

% Load clustering data for third step
clusters = cell(n_users,1);
clusters{1} = csvread('data/ClusterOne.csv');
clusters{2} = csvread('data/ClusterTwo.csv');
clusters{3} = csvread('data/ClusterThree.csv');
clusters{4} = csvread('data/ClusterFour.csv');

if print_user_input
    for i=1:n_users
        cluster = clusters{i}(:,2);
        for j=1:n_clusters
            mat = vec2mat(find(cluster == j), 7);
            printResultsWithImages(mat, rgbs);
        end
    end
end

for i=1:n_users
    cluster = clusters{i};
    scores(i,6) = getRandIndex(sys_c, cluster(:,2));
end
```

*******getNormalizedColorHistograms()*******

```
function [ hist ] = getNormalizedColorHistogram( img, sgmts )
% getColorHistogram takes in a color image and returns a 3D histogram made
% up of a number of bins (sgmts^3)
% img   = the 3D array representing RGB colors at each pixel of an image
% sgmts = the number of segments each color component (rgb) is broken into
% hist  = the histogram with pixel counts per bin

hist       = zeros(sgmts, sgmts, sgmts);
channels   = 255;
sgmt_len   = channels / sgmts;
img_size   = size(img);
sig_pixels = 0;

for i=1:img_size(1)
    for j=1:img_size(2)
        % Find rbg segments for pixels and increment their bin in hist
        % Max the value was 1 to handle cases where pixel value is 0
        rs = max(ceil(img(i, j, 1) / sgmt_len), 1);
        gs = max(ceil(img(i, j, 2) / sgmt_len), 1);
        bs = max(ceil(img(i, j, 3) / sgmt_len), 1);

        % Ignore pixels that are almost completely black - the "first" bin
        if rs > 1 || gs > 1 || bs > 1
            hist(rs, gs, bs) = hist(rs, gs, bs) + 1;
            sig_pixels = sig_pixels + 1;
        end
    end
end

hist = hist/sig_pixels;

end
```

*******l1Compare()*******

```
function [ similarity ] = l1Compare( vec1, vec2 )
%{
This function compares the normalized color vectors of two images and
returns a real number between 0 and 1 based on their similarilty.

vec1 = color vector for image 1
vec2 = color vector for image 2
similarity = degree of similarity (0 = not similar, 1 = perfectly similar)
%}

abs_diff = abs(vec1 - vec2);
similarity = 1 - (sum(abs_diff) / 2);

end
```

*******getSimilarityResults()*******

```
function [ results ] = getSimilarityResults( comps )
%{
Returns the results as specified by the assignment.

comps = 2D matrix that shows the similary comparisons for each image (could
be on colors or texture - values will be in the range [0, 1])

results = a 2D matrix of size (# of images x 7)
    - The first images is the image we are finding similarities against
    - The next three images are those that are the most similar, from most
    similar to third most similar
    - The last three images are those that are least similar, from third
    least similar to most least similar

Reference https://www.mathworks.com/matlabcentral/newsreader/view_thread/
248731 for sorting by a particular column.
%}

n = size(comps,1);
results = zeros(n, 7);

for i=1:n
    % Get similarities of other images with this image in ascending order
    img_cmps = [1:n; comps(i,:)];
    [~,I] = sort(img_cmps(2,:));
    img_cmps = img_cmps(:,I);

    results(i,1) = img_cmps(1,n);
    results(i,2) = img_cmps(1,n-1);
    results(i,3) = img_cmps(1,n-2);
    results(i,4) = img_cmps(1,n-3);
    results(i,5) = img_cmps(1,3);
    results(i,6) = img_cmps(1,2);
    results(i,7) = img_cmps(1,1);
end
```

*******getSimilarityGroup()*******

```
function [ results ] = getSimilarityGroup( cmps, opts )
% This function returns the most similar set of 4 images as described
% below. A set is ranked by their score, which is the sum of all pairwise
% comparisons within the set.
%
% cmps = the similarity comparisons already computer (NxN)
% results = the set of 4 images

findSimilar = true;
if (opts == Opts.Dissimilar)
    findSimilar = false;
end

N = size(cmps,1);
results = zeros(1,4);
score = 0;
if (~findSimilar) score = 4; end;

for i=1:N
    for j=(i+1):N
        for k=(j+1):N
            for l=(k+1):N
                imgs = [i j k l];
                current_score = getScore(cmps, imgs, findSimilar);
                if (findSimilar && current_score > score) ...
                        || (~findSimilar && current_score < score)
                    results = imgs;
                    score = current_score;
                end
            end
        end
    end
end
end

function score = getScore(cmps, imgs, findSimilar)
% Get the similarity score for the set of 4 images
% The score equals the sum of all pairwise comparisons of the set

score = 0;
N = length(imgs);
for i=1:(N-1)
    for j=(i+1):N
        score = score + cmps(imgs(i),imgs(j));
    end
end

end
```

*******printResultsWithImages()*******

```
function [ ] = printResultsWithImages( results, images)
%{
This function prints the results using the provided images. It formats the
prints to have five rows and the same number of columns as results.
%}

[rows, cols] = size(results);
for i=1:rows
    if (mod(i-1,5) == 0) figure(); end

    for j=1:cols
        if results(i,j) == 0
            continue;
        end;
        img = images{results(i,j)};
        pos = mod(i-1,5) * cols + j;
        subplot(5,cols,pos); subimage(img); axis off;
        title(results(i,j));
    end
end

end
```

*******getGrayScale()*******

```
function [ gray ] = getGrayScale( img )
% Returns the grayscale of an image
% Might want to try using MATLAB's rgb2gray() function

% OPTION 1
% gray = rgb2gray(img);

% OPTION 2
sz   = size(img);
imgD = double(img);
gray = zeros(sz(1), sz(2));

for i=1:sz(1)
    for j=1:sz(2)
        gray(i,j) = (imgD(i,j,1) + imgD(i,j,2) + imgD(i,j,3)) / 3;
    end
end

end
```

*******getLaplacian()*******

```
function [ result, max_val ] = getLaplacian( gray )
% This function takes in a grayscale image and returns its Laplacian
% Background pixels (grayscale < thresh) are assigned 0 as their value

sz = size(gray);
result = zeros(sz);
threshold = 37;
max_val = 0;

for i=1:sz(1)
    for j=1:sz(2)
        if gray(i,j) < threshold
            result(i,j) = -1;
            continue;
        else
            val = getValue(gray, i, j);
            result(i,j) = val;
            max_val = max(max_val, val);
        end
    end
end

end

function val = getValue(gray, row, col)
sz = size(gray);
val = 0;
n_neighbors = 0;

for i=(row-1):(row+1)
    for j=(col-1):(col+1)
        if i < 1 || i > sz(1) || j < 1 || j > sz(2)
            continue;
        else
            n_neighbors = n_neighbors + 1;
            val = val - gray(i,j);
        end
    end
end

val = val + (gray(row,col) * n_neighbors);
val = abs(val);
end
```

*******getNormalizedTextureHistogram()*******

```
function [ hist ] = getNormalizedTextureHistogram( laplacian, bins, max )
%{
This function will return the normalized texture histogram for a given
laplacian image.

laplacian = the laplacian image passed in
bins      = number of bins
hist      = the normalized texture histogram
%}

sz        = size(laplacian);
hist      = zeros(1, bins);
total     = 0;
bin_range = max / bins;

for i=1:sz(1)
    for j=1:sz(2)
        val = laplacian(i,j);
        if val ~= -1
            total = total + 1;
            if (val == 0)
                hist(1) = hist(1) + 1;
            else
                bin = ceil(val / bin_range);
                hist(bin) = hist(bin) + 1;
            end
        end
    end
end

hist = hist / total;

end
```

*******clusterSimilarities()*******

```
function [ result ] = clusterSimilarities( D, num_clust, option )
% D = distance matrix (N x N)
% num_clust = the desired number of clusters
% clustering = cluster labels for each index (N) of D

N = length(D);

% Initialize clusters
clusters = cell(N,1);
for i=1:N
    clusters{i} = i;
end

% Initialize cluster comparisons for memoization; -1 means not calculated
c_compares = ones(N,N);
c_compares = c_compares * -1;

while length(clusters) > num_clust
    C = length(clusters);
    dist = 1;
    c_to_join = [];
    for i=1:C
        for j=(i+1):C
            % Only computer if necessary
            if (c_compares(i,j) == -1)
                c_compares(i,j) = getDist(D, clusters{i}, clusters{j}, ...
                    option);
            end
            current_distance = c_compares(i,j);
            if (current_distance < dist)
                dist = current_distance;
                c_to_join = [i; j];
            end
        end
    end

    % Join the clusters into c_to_join(1), remove the second
    c1 = c_to_join(1);
    c2 = c_to_join(2);
    clusters{c1}   = [clusters{c1} clusters{c2}];
    clusters(c2,:) = [];

    % Flag c1 as needing to be compared again - remove c2
    c_compares(c1,:) = -1;
    c_compares(:,c1) = -1;
    c_compares(c2,:) = [];
    c_compares(:,c2) = [];
end

result = mapClusterArray(clusters, N);
```

```
end

function distance = getDist(D, c1, c2, option)
distance = 0;
if option == Opts.Single
    distance = 1;
end

n1 = size(c1,2);
n2 = size(c2,2);
for i=1:n1
    for j=1:n2
        current_dist = D(c1(i),c2(j));
        if option == Opts.Complete
            distance = max(distance, current_dist);
        else
            distance = min(distance, current_dist);
        end
    end
end
end

function map = mapClusterArray(cluster, N)
map = zeros(N,1);

for i=1:length(cluster)
    mat = cluster{i};
    for j=1:length(mat)
        map(mat(j)) = i;
    end
end

end
```

*******getScore()*******

```
function [ score ] = getScore( sys, usr )
% This function returns the system's score against the user input
% sys = (40 x 6) matrix - three most/worst similar ordered from 1 to 40
% usr = (40 x 2) matrix - users most and least similar ordered from 1 to 40
% score = (1 x 2) vector for most similar and most dissimilar scores

sim_score = 0;
dis_score = 0;
N = size(sys,1);

for i=1:N
    % Handle similar matches
    switch usr(i,1)
        case sys(i,1)
            sim_score = sim_score + 3;
        case sys(i,2)
            sim_score = sim_score + 2;
        case sys(i,3)
            sim_score = sim_score + 1;
    end

    % Handle dissimilar matches
    switch usr(i,2)
        case sys(i,6)
            dis_score = dis_score + 3;
        case sys(i,5)
            dis_score = dis_score + 2;
        case sys(i,4)
            dis_score = dis_score + 1;
    end
end

score = [sim_score, dis_score];

end
```

*******getRandIndex()*******

```
function [ ri ] = getRandIndex( c1, c2 )
% This function outputs the rand index between two clusters
% c1 = cluster 1
% c2 = cluster 2

N = length(c1);
total = 0;
agreed = 0;

for i=1:N
    for j=(i+1):N
        total = total + 1;
        if ((c1(i) == c1(j)) && (c2(i) == c2(j))) || ...
                ((c1(i) ~= c1(j)) && (c2(i) ~= c2(j)))
            agreed = agreed + 1;
        end
    end
end

ri = agreed / total;

end
```

*******Opts.m*******

```
classdef Opts
    % Some enumerated operations for use in the main script and functions
    enumeration
        Similar, Dissimilar, Complete, Single
    end
end
```