Melanie Hsu
mlh2197

**Note:** I added extra comments to some code snippets to give more context. Because of this, the code snippets in the report are more heavily commented than the code listed in the appendix.

**Citations:** I adapted code from this post by the user fraxel on how to concatenate images using PIL in Python: http://stackoverflow.com/questions/10647311/how-do-you-merge-images-using-pil-pillow

I also adapted code this fro this post by the user lalit on how to draw text onto an image: http://stackoverflow.com/questions/16373425/add-text-on-image-using-pil

Code in Stack Overflow can be used or adapted as long as the creator of the code is cited.

**Part 1: Gross Color Matching**
The python file used for this part is **color.py**. For this part, I used difference measures for most of the algorithms, but used similarity measures (1-difference) for the image table display and text file output.

In order to compare the three-dimensional color histograms of the forty images, I initialized a dictionary where each dictionary key is identified by a tuple (B, G, R), where B is the bin that the blue pixel belongs to, G is the bin that the green pixel belongs to, and R is the bin that the red pixel belongs to. The value for each key is how many pixels in the image fall into this (B, G, R) bin.

If I represented the RGB axes as full to 0 to 255 bins, the number of total bins in the histogram would be 255 cubed, or 16,581,375, and result in an average of 1 pixel per bin for each image. This would surely make image comparisons very difficult if not impossible, as we would end up comparing empty or very sparse bins against each other most of the time.

I decided to allocate eight bins to each of R, G, and B, resulting in a total of 512 bins. With a total of 89*60 = 5340 pixels per image, each bin in the image's histogram would have an average of 5340/512 = 10.43 pixels. Using just eight bins per color pixel thus greatly increases the chances that most of the 5K+ pixels in a given image will fall into many of the same bins that the 5K+ pixels of another image will.

Snippet 1 shows the code for generating a histogram for any given image, and Snippet 2 shows the rounding function used to organized the RGB pixels into bins. Each entry in the histogram, hist[(i, j, k)] basically holds a count of how many pixels in the image fall into the bin (B, G, R). For example, if the first pixel in an image was (255, 0, 255), then hist[(7, 0, 7)] would be incremented to 1 since we just saw a pixel belonging to the 8$^{th}$ blue bin, the 1$^{st}$ green bin, and the 8$^{th}$ red bin. If the second pixel was (240, 5, 247), then hist[(7, 0, 7)] would then be incremented to 2.

**Snippet 1**. Code for making a histogram for each image.

```
hist = {}#the initialized histogram
        img = Image.open(pict)#I stored the path of each picture in an array called pics
        pix = img.load()
        for i in range(0,9):#R, G, and B have 8 bins each, resulting in a histogram with 512 bins
                for j in range(0,9):
                        for k in range(0,9):
                                hist[(i, j, k)] = 0#initialize each histogram cell
        for i in range(0, 89):#iterate over the dimensions of the image
                for j in range(0, 60):
                        #print pix[i, j]
                        pixel = pix[i, j]#pixel[0] = B, pixel[1] = G, pixel[2] = R
                        #calculate the bin each pixel belongs to using my defined rounding function
                        blue = rnd(pixel[0])
                        green = rnd(pixel[1])
                        red = rnd(pixel[2])
```

```
                    #increment the histogram[(B, G, R)] count corresponding to the bins that the R,
                    #G, and B pixels belong to
                    hist[(blue, green, red)] += 1
        #counting is a variable corresponding to the number of the image
        picture[counting] = hist #store the histogram for this image in a dictionary
```

**Snippet 2.** Function for calculating the bin size.

```
#round each pixel value so that it fits into one of the bins, and return the bin number it belongs to
def rnd(x):
    base=32#since using R, G, B results in a lot of bins (8^3 in this case), I used a larger base
    return int(base*math.floor(float(x)/base))/base
```

After generating the histograms, I compared every image against every other image using my L1 norm function. In my L1 function, which is included in Snippet 3, I included logic to ignore "black" pixels under certain situations. All of the image comparisons essentially fall into three general cases:

Case 1: Image with black background vs. image with black background (ex. Image 23 vs. Image 24)
Because most of the images in our set have black backgrounds, it would make sense to ignore black pixels when comparing two images with black backgrounds. If we do not ignore the black pixels in this case, then the two images may seem more similar in color then they actually are due to a lot of their pixels falling into the first bin (the "black" pixel bin).

Case 2: Image with black background vs. image with full background (ex. Image 21 vs. Image 15)
When comparing an image with a black background against an image with a full or fuller background, ignoring the black pixels would cause the two images to seem more similar in color than they actually are. This would be a problem especially if the image with the black background has a central object that's similar in color to the contents of the image with the fuller background. If two images have very different backgrounds, then the similarity between them should be lower. Thus, in this case, we want to include the black pixels in the image comparison.

Case 3: Image with full background vs. image with full background (ex. Image 2 vs. Image 6)
In the third case of comparing two images with full or relatively full backgrounds, ignoring the relatively few black pixels in each image will not do much to affect the distance. Additionally, it's probably rare for two images with prominent non-black backgrounds to contain similar amounts of black pixels. One of the images may have more shadows or dark objects than the other, and that should be used to increase the calculated distance between the two images.

I decided to omit black pixels from the L1 norm calculations if two images had comparable numbers of pixels in the first bin. Specifically, if the difference between the counts in image1[0, 0, 0] and image 2[0, 0, 0] was less than 100 (I will refer to this in the remainder of the report as a "difference cutoff" of 100), the (0, 0, 0) bin of both images was omitted from the calculation. I chose a difference of 100 pixels based on my observations of the "black" pixel counts of the forty images. I noticed that images with prominent black backgrounds tended to have very similar black pixel counts.

My method for handling black pixels allowed for images with similar amounts of black pixels to be compared based on non-black colors, while subtracting similar numbers of pixels from consideration for each image. I didn't want to get into situations where I was subtracting 1000+ pixels from image1 (an image with a black background) and only 50-100 pixels from image2 (an image with a full background), which could potentially cause the color difference between the two images to be heavily underestimated.

**Snippet 3.** Function for calculating the L1 norm between two images

```
def L1norm(image1, image2):
    sum = 2*89*60#2N, where N = number of pixels in the image
```

```
    distance = 0
    #iterate through all of the (B, G, R) bins in the histogram
    for i in range(0, 8):
        for j in range(0, 8):
            for k in range(0, 8):
                #the pixels contained in the first R, G, and B bins are defined as “black”
                if i == 0 and j == 0 and k == 0:
                    #remove the black pixels from comparisons if the difference in black pixel
                    #count between the two images is small; if the images have very different
                    #counts of black pixels then the pictures should be rated as very different
                    #from each other
                    if math.fabs(image1[0,0,0] - image2[0,0,0]) < 100:
                        #if we remove black pixels from comparisons, we must subtract
                        #the number of black pixels in these bins from the total sum
                        sum -= image1[0,0,0]
                        sum -= image2[0,0,0]
                        continue
                distance += math.fabs(image1[i,j,k] - image2[i,j,k])
    norm = distance/sum
    return norm
```

Based on the L1 norm calculations, I had color.py generate a dictionary of dictionaries, containing information about each image's similarity to every other image. Below is the dictionary entry for Image 1, which I reformatted and output to a text file, colortable.txt. The text in red was added in for clarity, and the red numbers indicate the number of the image that Image 1 is being compared against.

{1:1.0 2:0.34363 3:0.62453 4:0.53483 5:0.4 6:0.14176 7:0.25 8:0.5309 9:0.28202 10:0.67771 11:0.21779 12:0.14794 13:0.29532 14:0.30431 15:0.02135 16:0.58858 17:0.24981 18:0.26536 19:0.53558 20:0.29213 21:0.35543 22:0.40506 23:0.25056 24:0.4882 25:0.29307 26:0.05674 27:0.33352 28:0.27903 29:0.42622 30:0.32172 31:0.22828 32:0.35506 33:0.26779 34:0.35506 35:0.26049 36:0.51255 37:0.38277 38:0.52959 39:0.4103 40:0.29944}

As expected, Image 1 has a similarity value of 1.0 with itself. From this dictionary entry, we can see that Images 10 (0.67771), 3 (0.62453), and 16 (0.58858) are most similar to Image 1, and Images 6 (0.14176), 26 (0.05674), and 15 (0.02135) are most different from Image 1. Snippet 4 shows the code for calculating the three most similar and different images for each image.

**Snippet 4.** Code for calculating the three most similar and different images for each image

```
#each image has a septuple consisting of the image, the three images most similar to it, and the three
#images most different from it
sept = {}
    for i in range(0, 40):#initialize a 2D array
      sept[i+1] = {}#I started at the 1st index since the first image is Image 1, to avoid confusion

    #compute the three most similar and most different images
    for i in range(len(picture)):
      topfoe = I #the most different image
      twofoe = I #the second most different image
      threefoe = I #the third different image
      one = 0
      two = 0
      three = 0
```

```
topfriend = I #the most similar image
twofriend = I #the second most similar image
threefriend = I #the third most different image
yi = 1
er = 1
san = 1
for j in range(len(picture)):#find the two most similar and two most different images
        #logic to find the two most different images from each image
        if differences[i+1][j+1] > one:
                two = one
                twofoe = topfoe
                one = differences[i+1][j+1]
                topfoe = j+1
        elif one > differences[i+1][j+1] > two:
                two = differences[i+1][j+1]
                twofoe = j+1
        #logic to find the two most similar images to each image
        if 0 < differences[i+1][j+1] < yi:
                twofriend = topfriend
                er = yi
                topfriend = j+1
                yi = differences[i+1][j+1]
        if yi < differences[i+1][j+1] < er:
                twofriend = j+1
                er = differences[i+1][j+1]
sept[i+1][0] = i+1
sept[i+1][1] = topfriend
sept[i+1][2] = twofriend
sept[i+1][5] = twofoe
sept[i+1][6] = topfoe

for j in range(len(picture)):#find the third most similar and third most different image
        #don't double-count the most different and second most different images
        if differences[i+1][j+1] > three and j+1 != topfoe and j+1 != twofoe:
                three = differences[i+1][j+1]
                threefoe = j+1
        #don't double-count the most similar and second most similar images
        if 0 < differences[i+1][j+1] < san and j+1 != topfriend and j+1 != twofriend:
                san = differences[i+1][j+1]
                threefriend = j+1
sept[i+1][3] = threefriend
sept[i+1][4] = threefoe
```

After finding the three most similar and three most different images for each image, I used OpenCV's image concatenation functions to make a pictorial representation of the septuples for each image.

**Figure 1**. Table of septuples for the forty images, based on color similarity measures. I split the table into two length-wise for readability purposes.

For each row:
Col. 1: the image
Col. 2: the image most similar to Col. 1
Col. 3: the 2nd most similar image to Col. 1

Melanie Hsu
mlh2197

Col. 4: the 3rd most similar image to Col. 1
Col. 5: the 3rd most dissimilar image from Col. 1
Col. 6: the 2nd most dissimilar image from Col. 1
Col. 7: the image that is most different from Col. 1

The number in the top left of each thumbnail represents the number of the image. The number in the bottom left of each thumbnail represents the similarity value between the image in Col. 1 and the image in Col. n, where n = 1, 2, 3, 4, 5, 6, or 7.

Melanie Hsu
mlh2197

Overall, the color comparisons seem pretty reasonable. Most of the three similar images for each image were on the same side of the color wheel as the image (as they should be), while the three most different images were either on the opposite side of the color wheel (ex. Image 20's foreground object is reddish-pink, and its most different image is Image 29, whose foreground object is whitish-green) or differed notably from the image in its dominant color (ex. the green and blue fractal in Image 32 vs. the predominantly pinkish-red Image 15).

It seems that my method of dealing with black pixels worked pretty well when it came to finding the three most similar images. The decision to not omit the first bin in cases where two images had significantly different amounts of black pixels but were otherwise very similar in color probably caused their similarity values to be lowered more than necessary. For instance, Image 2 may have had higher color similarity with Image 39 had the black background of the latter image been removed from the calculations. However, this did not cause any major problems, as a visual inspection shows that most images are very similar in color to their three most similar images, the exception being images that are very different in color from everything else (ex. the predominantly purple and black Image 31).

The black pixel handling method did not work as well for the three most different images, the similarity values for the most different images to be very low. As an example, I believe that Image 37's top foe should have been Image 35 instead of Image 25. However, both Image 35 and 25 have orange and black as their predominant colors, so my algorithm was probably not too far off anyways. It should also be noted that Image 35 has a similarity value of 0.00898 with Image 37, while Image 25 has a similarity value of 0.00403 – a tiny difference. Tweaking the black pixel handling method slightly would probably cause a reordering of some of the three most different images.

I decided to keep my original black pixel handling parameters for the following reasons:

1) Finding the most different image(s) is, in my opinion at least, a more difficult and less precise task compared to finding the most similar image(s). In real life, people tend to agree more on "most similar color" than on "most different color" – this was supported by the results of my user study.
2) The similarity between two images of opposite colors should be very close to 0, if not 0. This is supported by the fact that Image 17, whose foreground object is red, orange, and yellow, and Image 30, whose foreground object is green and blue, are not only each other's most different images, but also have a similarity score of 0.0. Many other image pairs with near-opposite colors have similarity scores that are very close to 0.
3) As an experiment, I tried adjusting the black pixel handling method (ex. using a cutoff of 150, 200, or 250 instead of 100) and put the results in Figure 1b. It seems that using a difference cutoff of near 100 is optimal for color similarity, and deviating too far from this number lowers the accuracy of the top three images. Tweaking the cutoff sometimes changes the similarity scores enough to cause an reordering of the three most similar and different images.

**Figure 1b.** The effects of dealing with black pixels in other ways.

From left to right, the images represent the effects of the following experiments:
Never ignore black pixels: causes some of the three most difference measures to deteriorate.

Remove the first bin for every image: the effects of this is slight, but there is a degradation in the quality of most similar images (ex. Image 13's most similar image shouldn't be Image 10).

Use a difference cutoff of 50: is actually better than using a cutoff for 100 if user agreement is any indication, as it raises user agreement with both color similarity (slightly, less than 1% each) and color

difference (1-3% rises each). However, the four most different images includes a greenish-yellow fruit and a green vegetable – they definitely shouldn't be in this group together. See Figure 2a.

Use a difference cutoff of 150: reasonable, but barely different from using a cutoff of 100 - user agreement with color similarity falls by 1% on average, but agreement with color difference rises by 1%.

Use a difference cutoff of 250: too extreme: a vegetable should never have a fractal as one of its top three most similar images, as there are so many green veggies in the set that any fractal should never become a top three candidate. But Image 37 has two fractals as its second and third most similar images.

**Figure 1c.** Four most different images, if I used a difference cutoff of 50 instead of 100. This is why I chose not to use a cutoff of 50 pixels in my method for handling black pixels.

Finally, from a first glance, Image 15 seems to be the most frequent "top foe," or the most different image for many of the images (18/40) and also appeared as the second or third most different image for many images. This was later confirmed when Image 15 appeared as one of the four most different images.
In order to find the four most similar and different images, I compared every possible group of four that you could form from the forty images against every other possible group of four. I stored the results of all of these comparisons in a dictionary, where each key in the dictionary, (A, B, C, D), represents four different images and the value for the key represents the average of the similarity values between the four images. This average was obtained by adding the differences between every two images in the quartet (ex. A vs. B, A vs. C, A vs. D, B vs. C, B vs. D, and C vs. D), and dividing this value by 6 (the number of comparisons made). The logic of doing these comparisons is that if A, B, C, and D are very similar to each other, then this average distance would be small (high similarity); conversely, if they are very different from each other, this average distance would be large (low similarity).

**Snippet 5.** Code to compare every group of four in the set of forty images.

```
compares = {}
for i in range(len(differences)):
   for j in range(len(differences)):
            for k in range(len(differences)):
                     for l in range(len(differences)):
                              #don't compare any image to itself
                              if i != j and i != k and i != l and j != k and j != l and k != l:
                              #average the difference between each image to every other image
                                       average = (differences[i+1][j+1] + differences[i+1][k+1] +
                                                differences[i+1][l+1] + differences[j+1][k+1]
                                                + differences[j+1][l+1] + differences[k+1][l+1])/6
                                                compares[(i+1,j+1,k+1,l+1)] = average
```

Having stored all of the group similarity information in a dictionary, I now search through the dictionary to find the group of four images with the highest and lowest average difference (lowest and highest average similarity). The code for finding the most similar and most different group is pretty similar, so I included only the code for finding the four most different images.

**Snippet 6.** Code to find the four most different images

```
#find the four images that are most different from each other
tuple = (i,j,k,l)
biggest = 0
bigtuple = (0,0,0,0)
for tuple in compares:#compares already contains tuples of four images in each cell
   #at the end of this for loop, we will have found the tuple corresponding to the four most diff images
    if compares[tuple] > biggest:
            biggest = compares[tuple]
            bigtuple = tuple
```

**Figure 2**. The four most similar images, according to color similarity measures.


The four most similar images are green vegetables. This isn't a surprise, as most of the green vegetables in the dataset have other green vegetables as their three most similar images. This group has an average similarity score of 0.727, which is pretty high considering that the two most similar images in the dataset have a similarity score of 0.79513.

These four images all lie in the green to greenish-yellow part of the color wheel, providing support to them being the four most similar images in the dataset. Interestingly, only one of the three users placed all four of these images in the same cluster, showing that users won't necessarily categorize images similarly based on just color (and for good reason too, since there are a lot of other things that people look for in similar images, such as texture and shape – ex. stringy or round foods).

**Figure 3**. The four most different images, according to color similarity measures.


This is also not too surprising as four of these images were the "top foe" to at least two images, accounting for 25/40 of the "top foes" in the data set. Images 16 and 31 are also the "top foes" of each other, with a similarity score of less than 0.005. The average similarity score in this group is 0.035805.

These four images almost divide the color wheel into four equally-sized quarters, with Image 16 in the green to yellowish-green zone, Image 12 in probably around the orange zone, Image 15 in the reddish-pink zone, and Image 31 in the bluish-purple zone. This lends support to these four images being the most different in the set, color-wise.

Two of the three users placed these four images in different clusters, suggesting that color was a major consideration for them in categorizing these images.

**Part 2: Gross Texture Matching**

The python file for this part is **texture.py**. Like with Part 1, I used difference measures for most of the algorithms, but used similarity measures (1-difference) for the image table display and text file output.

**Snippet 7.** Code for making a grayscale image. I used OpenCV to build the image pixel by pixel.

```
#make a grayscale image for each image
    for pict in pics:
        gray = {} #the grayscale image will be saved in this dictionary, pixel by pixel
        img = Image.open(pict)
        pix = img.load()
        grayimg = Image.new('L', (89, 60)) #make a new grayscale image
        for i in range(0, 89):#iterate over every pixel in the picture
                for j in range(0, 60):
                        gray[i, j] = 0
                        pixel = pix[i, j]
                        gray[i,j] = (pixel[0] + pixel[1] + pixel[2])/3 #(R + G + B)/3 to get grayscale pixel
                        grayimg.putpixel((i, j), gray[i,j]) #build the image pixel by pixel
```

After creating and saving a grayscale image, I loaded the image back in and built a Laplacian image from it using the code in Snippets 8, 9, and 10. From printing out all of the Laplacian pixel values and finding the minimum and maximum value using Microsoft Excel, I discovered that the Laplacian pixel values ranged from -1320 to 1500, covering a range of nearly 3000 values. Since each image has 5340 pixels, using 3000 bins is unreasonable, since that would mean that each bin will theoretically contain an average of under 2 pixels in the best case. In reality, most bins would be empty, as a large proportion of pixels will probably be concentrated near 0 for the images that feature smooth objects.

I decided to use 30 bins, where each bin covers a range of 100 Laplacian pixel values, allowing each bin to theoretically contain an average of 178 pixels. Since the pixel values can vary wildly both within and

between images, using more bins would probably decrease the likelihood that the 5K+ pixels of two images will fall in the same bins. In my bin sorting function, I specifically added 1320 to each Laplacian pixel value so that the pixels would range from 0 to 2820, making them easier to sort into bins. This correction was made specifically for this dataset, and would not work for datasets where one or images have a pixel(s) with the minimum Laplacian value of -2040, the case where a pure black pixel is surrounded by eight pure white pixels.

**Snippet 8.** Function to sort the Laplacian pixels into bins.

```
def sort(x): #range: -1320 to 1500, sort the Laplacian pixels into bins
        base=100
          return int(base*round(float(x+1320)/base))/base #adjust so that smallest value = 0
```

Before making the Laplacian image, I checked each grayscale pixel to see whether it would fall into the first bin. Based on a visual inspection of the grayscale colors, I decided to define the first thirty-two pixels as “black.” Since the program will only remove a pixel from the L1 norm calculations if it is designated as black, we don’t really care about what bins the non-black bins fall into. Thus, I used a large base of 32, which divided the grayscale axis into 8 bins, to check for black pixels.

**Snippet 9.** Function to sort the grayscale pixels into bins. Used to check for “black” pixels.

```
#round each grayscale pixel value so that it fits into a bin, and return the bin number it belongs to
def rnd(x): #we only really care about whether the pixel is “black” or not
        base=32
        print x
          return int(base*round(float(x)/base))/base
```

If the grayscale pixel fell into the first bin, then I did not convert it into a Laplacian pixel. Otherwise, if the pixel wasn’t designated as black, it was converted into a Laplacian pixel using the code in Snippet 10.

The series of if-statements starting from “if j > 0:” helps to determine the number of neighbors that each pixel has. Each pixel can have as few as 3 neighbors, if it’s located in one of the corners of the image:

```
[ x ][   ][   ]      [   ][   ][ x ]      [   ][   ][   ]      [   ][   ][   ]
[   ][   ][   ]      [   ][   ][   ]      [   ][   ][   ]      [   ][   ][   ]
[   ][   ][   ]      [   ][   ][   ]      [ x ][   ][   ]      [   ][   ][ x ]
```

If the pixel is along the edge of the image, but not in one of the four corners, then it will have 5 neighbors:

```
[   ][ x ][   ]      [   ][   ][   ]      [   ][   ][   ]      [   ][   ][   ]
[   ][   ][   ]      [   ][   ][ x ]      [   ][   ][   ]      [   ][   ][   ]
[   ][   ][   ]      [   ][   ][   ]      [   ][ x ][   ]      [   ][   ][ x ]
```

Finally, if the pixel is neither along the edge or corner of the image, it will have 8 neighbors:

```
[   ][   ][   ]
[   ][ x ][   ]
[   ][   ][   ]
```

The logic in Snippet 10 takes care of all of the above scenarios, incrementing the multiplier, which represents the number of neighbors, by 1 for each of the pixel’s neighbors, and subtracting each neighbor’s pixel value from the multiplier*the pixel’s grayscale value. The result of these operations is the pixel’s Laplacian value, which is stored in the appropriate bin of the histogram.

Melanie Hsu
mlh2197

**Snippet 10.** Code for making a Laplacian image. The image was already loaded in and the grayscale pixel array for the image has been obtained.

```
hist = {}#histogram holder
for i in range(0, 30): #initialize a dictionary
        hist[i] = 0
for i in range(0, 89):
    for j in range(0, 60):
        lap[i, j] = 0
        pixel = pix[i, j] #get the grayscale pixel
        rounded = rnd(pixel) #round to see which bin the grayscale pixel falls into
        if rounded == 0: #don't calculate laplacian value for “black” pixels
            black[int(string)] += 1
            continue
        multiplier = 0 #number of neighbors the pixel has
        subtractor = 0 #how much to subtract from 8*the pixel’s value
        if j > 0:
            multiplier += 1#i, j-1 doable
            subtractor += pix[i, j-1]
        if j < 59:
            multiplier += 1#i, j+1 doable
            subtractor += pix[i, j+1]
        if i > 0:
            multiplier += 1#i-1, j doable
            subtractor += pix[i-1, j]
            if j > 0:
                multiplier += 1#i-1, j-1 doable
                subtractor += pix[i-1, j-1]
            if j < 59:
                multiplier += 1#i-1, j+1 doable
                subtractor += pix[i-1, j+1]
        if i < 88:
            multiplier += 1#i+1, j doable
            subtractor += pix[i+1, j]
            if j > 0:
                multiplier += 1#i+1, j-1 doable
                subtractor += pix[i+1, j-1]
            if j < 59:
                multiplier += 1#i+1, j+1 doable
                subtractor += pix[i+1, j+1]
        #laplacian value = # of neighbors*original pixel val – pixel val of each neighbor
        lap[i, j] = multiplier*pixel - subtractor
        bin = sort(lap[i, j]) #sort the calculated laplacian value into a bin
        hist[bin] += 1 #increment the count for the appropriate histogram value
```

I basically built the texture histogram using the same technique that I used in Part 1, except that I use 30 bins instead of 512, and the histogram is one-dimensional. Each pixel in the image will fall into one of the 30 bins in the histogram, and depending on which bin the Laplacian pixel belongs to, the count of the appropriate bin in the histogram will be incremented. For instance, if the first pixel has a Laplacian value of 1300, then it falls into the 26$^{th}$ bin, and hist[25]’s count will be incremented to 1. If the second pixel has a Laplacian value of -5, then it falls into the 13$^{th}$ bin, and hist[13]’s count will be incremented to 1.

Once we make the Laplacian image, there’s no way to tell which Laplacian values correspond to black pixels, since the Laplacian values are a reflection of how textured the image is. However, because none

of my three users factored in the black pixels when making their texture comparisons, I decided to ignore the black pixels in every single image. Thus, before calculating the histogram, I made an array whose cells represent the number of pixels designated as black in an image.
Unlike with Part 1, I subtracted the black pixel counts in both images from the sum, regardless of whether the two images have similar amounts of black pixels. Previously, I removed the black pixel counts of both images from the sum only if their counts differed by less than 100. Because both implementations (subtract from all images vs. subtract only from images with similar black pixel counts) yielded more or less the same levels of user agreement, I went with subtracting the black pixel counts of all images from the sum. This is actually a more reasonable technique, since we never had the black pixel Laplacian values in the histograms to begin with. If I had done subtractions under some (but not all) situations, then for those image comparisons where black pixel counts weren't subtracted from the sum, I would be dividing the distance by too large of a sum. This would've caused an artificial lowering of some of the difference values (and therefore an artificial raising of some of the similarity values).

**Snippet 11.** Function that calculates the L1 norm between every pair of images.

```
#calculate the L1_Norm between every pair of images
def L1norm(image1, image2, black1, black2):
        sum = 2*89*60
        distance = 0

        #subtract the black pixel count for both images from the sum
        sum -= black1
        sum -= black2
        for i in range(0, 30):
           distance += math.fabs(image1[i] - image2[i])
        norm = distance/sum
           return norm
```

**An alternate way of handling black pixels:**

At one point, I considered taking the Laplacian values of the black pixels into consideration in cases where the black pixel counts of the two images did not differ by less than 100.

In this situation, I provided the histograms of the two images being compared, two histograms that include the Laplacian values of all black pixels, and the black pixel counts of the images to the L1 norm function.

**Alt 1.** L1 norm function that uses different histograms based on whether the number of black pixels in each image differs by less than 100. If the difference is greater than 100, then black pixels cannot be ignored, and we must compare the image histograms that include the Laplacian values of all black pixels.

```
#calculate the L1_Norm between every pair of images
def L1norm(image1, image2, blackhist1, blackhist2, black1, black2):
        sum = 2*89*60
        distance = 0

        #subtract the black pixels from every image
        if math.fabs(black1 - black2) < 100:
           sum -= black1
           sum -= black2
        else:#use the histogram that includes black pixels instead
           image1 = blackhist1
           image2 = blackhist2
        for i in range(0, 30):
```

```
    distance += math.fabs(image1[i] - image2[i])
  norm = distance/sum
    return norm
```

**Alt 2.** Alternate code for making a Laplacian image, including a backup histogram containing the Laplacian values of all pixels, including the ones designated as black pixels.

```
  picarray = {}
  picbackup = {}  #holds a backup histogram that includes black pixels
  black = {} #number of pixels that will be discarded as black in each image
  #(need it to know how much to subtract when doing L1 calcs)
  for i in range(1, 41):
    black[i] = 0 #initialize the black pixel count
  for picture in graypics:
    lap = {}
    blackpixlap = {} #laplacian image that includes black pixels
    hist = {} #histogram holder
    blackpixhist = {} #histogram holder that also includes black pixels
    string = graypics[count].split("/")[-1]
    string = string.replace("gray.ppm", "")
    img = Image.open(picture)
    pix = img.load()
    for i in range(0, 30): #initialize histogram
                        hist[i] = 0
                        blackpixhist[i] = 0
    for i in range(0, 89):
              for j in range(0, 60):
                        lap[i, j] = 0
                        blackpixlap[i, j] = 0
                        pixel = pix[i, j]#grayscale pixel
                        rounded = rnd(pixel)#round to see which bin the grayscale pixel falls into
                        if rounded == 0:
                                  black[int(string)] += 1
                        multiplier = 0
                        subtractor = 0
                        if j > 0:
                                  multiplier += 1#i, j-1 doable
                                  subtractor += pix[i, j-1]
                        if j < 59:
                                  multiplier += 1#i, j+1 doable
                                  subtractor += pix[i, j+1]
                        if i > 0:
                                  multiplier += 1#i-1, j doable
                                  subtractor += pix[i-1, j]
                                  if j > 0:
                                            multiplier += 1#i-1, j-1 doable
                                            subtractor += pix[i-1, j-1]
                                  if j < 59:
                                            multiplier += 1#i-1, j+1 doable
                                            subtractor += pix[i-1, j+1]
                        if i < 88:
                                  multiplier += 1#i+1, j doable
                                  subtractor += pix[i+1, j]
                                  if j > 0:
```

```
                        multiplier += 1#i+1, j-1 doable
                        subtractor += pix[i+1, j-1]
                if j < 59:
                        multiplier += 1#i+1, j+1 doable
                        subtractor += pix[i+1, j+1]
        blackpixlap[i, j] = multiplier*pixel - subtractor
        if rounded != 0:#only store the non-black pixels in this dictionary
                #each pixel value is 8*original pixel value
                lap[i, j] = multiplier*pixel - subtractor
                bin = sort(lap[i, j])#calculate the bin for the Laplacian pixel
                hist[bin] += 1
        elif : #store every pixel in this dictionary, whether “black” or not
                bin = sort(blackpixlap[i, j])
                blackpixhist[bin] += 1
picarray[int(string)] = hist #store the histogram for the image
picbackup[int(string)] = blackpixhist
        count += 1
```

This approach ended up hurting my user agreements more than it helped, so I ended up discarding it. Specifically, User 1 had a texture similarity score of 0.857232084155 and a difference score of 0.900049309665, User 2 had a similarity score of 0.818425378041 and a difference score of 0.793507560815, and User 3 had a similarity score of 0.854569362262 and a difference score of 0.841962524655. On average, the three users had a similarity score of 0.843408941 and a difference score of 0.845173132.

Compared to the 0.818293886 and 0.905506246 similarity and differences scores that I got with my other implementation, this alternate implementation increased the similarity score by 0.025115056, but crashed the difference score by 0.060333114. Overall, this alternate method produced poorer agreements.

It also turns out that this alternate approach isn't that great of an idea from a user standpoint either, because none of the three users I asked made image texture rankings based on the textures of the black pixels in the images. Instead, they looked at the foreground object in the image (if the image had a black background), or at the entire image (if the image had a full background).

In light of the results of this experiment, I decided to stick with my original way of handling black pixels.

Besides increasing the number of arguments provided to the L1 norm function, I pretty much used the same code from Part 1 for the rest of this section. The texture dictionary, which was output to a text file, texturetable.txt, works just like the color dictionary. I added in red text again for clarity, and the red numbers indicate the number of the image that Image 1 is being compared against.

{1:1.0 2:0.55179 3:0.89388 4:0.78846 5:0.7343 6:0.54797 7:0.84622 8:0.88815 9:0.86806 10:0.94352 11:0.78018 12:0.60236 13:0.73639 14:0.60985 15:0.66076 16:0.7757 17:0.69761 18:0.58788 19:0.75111 20:0.54259 21:0.51587 22:0.50577 23:0.69338 24:0.69648 25:0.59225 26:0.23988 27:0.69993 28:0.63391 29:0.45218 30:0.46707 31:0.80807 32:0.58987 33:0.57758 34:0.53832 35:0.44157 36:0.5553 37:0.46024 38:0.68219 39:0.68546 40:0.80323}

As expected, Image 1 has a 1.0 similarity with itself, and judging from the dictionary entry for Image 1, the three most texturally similar images are Images 10 (0.94352), 3 (0.89388), and 8 (0.88815). The three most different images for Image 1 are Images 29 (0.45218), 35 (0.44157), and 26 (0.23988). This and the septuples for the other images in the dataset are shown in Figure 4.

Melanie Hsu
mlh2197

The code and rationale behind the three most similar and different image calculations, and the code and rationale for calculating the four most similar and different images, are virtually identical to Part 1. Because of this, I will spend the rest of this section discussing my results.
Figure 4 shows the septuples for each image. Like with Part 1, one image, Image 26, stood out as the "top foe" for the majority of images in the set. For texture, however, the homogeneity of the "top foe" images was even more profound. Literally every image in the set was had either Image 26 (32/40), the picture of a sunset, or Image 8 (8/40), the picture of the asparagus, as their most different image.

**Figure 4**. Table of septuples for the forty images, based on texture similarity measures. I split the table into two length-wise for readability purposes. Row and columns are organized the same was as Figure 1.

Melanie Hsu
mlh2197

The texture output is very good. When examining the grayscale versions of images, I found that Image 26 is much smoother than most of the other images, filled with only gradual spikes in image intensity. Its closest match, Image 30, is almost as smooth. This match isn't surprising from a common-sense perspective either, given that the texture of air (Image 26) is different from the texture of concrete, solid objects, which are found in most of the images. Since that the majority of images also have moderate to very sharp local changes, it's expected that Image 26 will be the most frequent "top foe."

The softer single fruits and vegetables, such as Image 20 and Image 36, which have smoother and more gradual intensity changes, were matched as most similar to each other. In contrast, the rough single fruits and vegetables, such as Image 19, which has many abrupt local changes due to the center of the wiki, are consistently paired with other roughly-textured foods, such as Image 23. It may not look like it, but Image 23 has many abrupt local changes due to the spokes of the tomato.

As for why the texturally softer fruits had both the texturally roughest images (ex. Images 1 and 8) and the incredibly smooth Image 26 among their three most different images, it's because the fruits were intermediate in terms of textural roughness. Because of this, both the very rough and very smooth images are considered to be very different from them.

Interestingly, the texture similarity values are overall much higher than the color similarity values (the two most similar images, Images 30 and 37, have a texture similarity of 0.96306, while the two most different images, Images 26 and 8, have a similarity of 0.2189). Based on these numbers, I expect 30 and 37 to be two of the four most similar images, and for Images 26 and 8 to be two of the four most different images. This was shown to be true.

The four most similar images had an average group texture similarity value of 0.93769, which is very high. In examining the grayscale images, I found that all four images are fairly smooth, especially Images 30 and 22. Except for the places where the most vivid colors meet the black background, the images consisted of mostly gradual local changes.

**Figure 5.** The four most similar images, according to texture similarity measures.


The four most different images have an average group texture similarity value of 0.44425666666. The drastic textural difference between Images 26 and 8 was probably softened by Image 2, which is intermediate in smoothness, with more drastic local changes at the bottom of the image and smoother changes at the top. Image 33, whose local changes are moderate (a lot of black going to gray, and gray going to white), also helped to raise the similarity between the four images.

**Figure 6**. The four most different images, according to texture similarity measures.


Based on these observations and results, it seems that the algorithm did a great job of sorting images based on texture, and finding the four most similar and different images.

Figure 7 shows what would be the four most similar and different images if I had only subtracted black pixels for images with similar backgrounds. Image 8 and 10 are clearly very similar in texture (in fact, 10 is 8's closest match), and therefore should not be in this cluster together. Therefore, my decision to subtract

black pixels from all images clearly resulted in a better output, even though the user agreement between these two implementations was very similar.

**Figure 7.** Most similar (top) and different (bottom) images for an alternate black pixel handling method.


**Part 3: Colors and Textures**

The python file for this part is **colorxtexture.py**.

In preparation for the clustering algorithm and linear sum calculations, I had my program read in text file versions of the color and texture dictionaries that I made in Parts 1 and 2 (colortable.txt and texturetable.txt), and convert them back into dictionary form. As mentioned in Part 1, the color and texture dictionary for each image contains 40 similarity values, obtained by comparing the image against every image in the dataset, including itself.

I then calculated the linear sum (S) between every pair of images, feeding the color and texture similarity values between the images being compared to a linear sum function that I defined. This allowed me to combine the color and texture similarities between each pair of images into a single similarity value. The contribution weights of color and texture are determined by the value of r, where r = 1.0 signifies a pure texture comparison, and r = 0.0 signifies a pure color comparison.

I stored the distance, 1 – S, in each entry in the linsum dictionary.

**Snippet 12.** Calculating the linear sum between every pair of images.

```
linsum = {}
for i in range(0, 40):#initialize a 2D array to store the distance measures (1-linear sum)
   linsum[i+1] = {}
for i in range(0, 40):
   for j in range(0, 40):
                        linsum[i+1][j+1] = 1-(linearsum(colors[i+1][j+1], textures[i+1][j+1]))
```

**Snippet 13.** Function for calculating the linear sum.

```
#calculate the linear sum of an image
def linearsum(color, texture):
    #if you change the r-value, re-run colorxtexture.py, then run user.py (part 4 code), you can
    #immediately see the effects of the r-value you used on user agreement scores, since
    #user.py outputs numbers corresponding to the user recall and precision
    r = 0.5
    linsum = r*float(texture) + (1.0-r)*float(color)
      return linsum
```

I chose the r-value based on a combination of user agreement and how reasonable the clusters looked. Since both my users and myself had more trouble comparing images by texture, it was expected that my clusters would be not match well with the clusters made by the three users if I set the r-value too high,

heavily favoring texture in the linear sum equation. As expected, when I adjusted the r-value to 0.7 or higher, then calculated the resulting user agreement levels using my part 4 code, user.py, the recall and precision tended to suffer for all three users.

The values near r = 0.5 yielded the highest levels of user agreement, which is expected since people tend to use a combination of color and texture to group images. Average recall for the three users was highest when r = 0.6, while average precision was highest when r = 0.4. r = 0.5 gave the 2nd best recall and 3rd best precision; however, the gap between the best and 2nd best average recall was 0.110648148, a huge drop. To decide on a final r-value, I decided to compare the clusters generated when r = 0.4, 0.5, and 0.6.

**Table 1.** User cluster precision and recall for different r values. I decided to use increments of 1.0, as changes in r of 0.75 or less did not result in any changes in recall or precision.

| R-value | Recall/precision (User 1) | Recall/precision (User 2) | Recall/precision (User 3) |
|---|---|---|---|
| 0.0 | 0.418333333333/ 0.680934343434 | 0.34/ 0.598333333333 | 0.365833333333/ 0.575 |
| 0.1 | 0.418333333333/ 0.680934343434 | 0.34/ 0.598333333333 | 0.365833333333/ 0.575 |
| 0.2 | 0.416176470588/ 0.593434343434 | 0.358137254902/ 0.573333333333 | 0.368921568627/ 0.56 |
| 0.3 | 0.374350649351/ 0.571843434343 | 0.414826839827/ 0.698333333333 | 0.405303030303/ 0.61 |
| 0.4 | 0.382575757576/ 0.61351010101 | 0.414393939394/ 0.708333333333 | 0.405303030303/ 0.62 |
| 0.45 | 0.382575757576/ 0.61351010101 | 0.414393939394/ 0.708333333333 | 0.405303030303/ 0.62 |
| 0.5 | 0.406944444444/ 0.528787878788 | 0.472222222222/ 0.705 | 0.434722222222/ 0.62 |
| 0.6 | 0.535416666667/ 0.530303030303 | 0.559375/ 0.678333333333 | 0.551041666667/ 0.566666666667 |
| 0.7 | 0.42196969697/ 0.480176767677 | 0.44196969697/ 0.524583333333 | 0.413787878788/ 0.434583333333 |
| 0.8 | 0.35/ 0.463636363636 | 0.466666666667/ 0.668333333333 | 0.402777777778/ 0.541666666667 |
| 0.9 | 0.364383394383/ 0.344444444444 | 0.511111111111/ 0.496666666667 | 0.41242979243/ 0.408333333333 |
| 1.0 | 0.37371031746/ 0.35101010101 | 0.459861111111/ 0.455 | 0.405218253968/ 0.388333333333 |

R = 0.4 doesn't seem like the best choice, as it resulted 4 out of 7 clusters having only one image. The single-image clusters themselves aren't reasonable, as it's highly unlikely that users would put Image 15, which is in the same color region as Images 5, 6, 7, 9, and 11 in its own cluster. Content-wise, there are many images of clustered fruits and vegetables in the database, so it's highly unlikely for users to not bunch up Image 15 with at least a few other images of this kind.

There are a few major differences between the clusters created with r = 0.5 and r = 0.6. First, for r = 0.5, the red and green clustered vegetables are placed in a single cluster, and for r = 0.6, these vegetables are placed in separate clusters. Another notable difference is that for r = 0.5, Images 27 and 31, two fractals, are placed in their own cluster, while for r = 0.6, two fractals, Images 27 and 32, are grouped with a cheese (Image 34). Lastly, for r = 0.6, the purple fractal, Image 31, is grouped with the images featuring clusters of red fruits and vegatables.

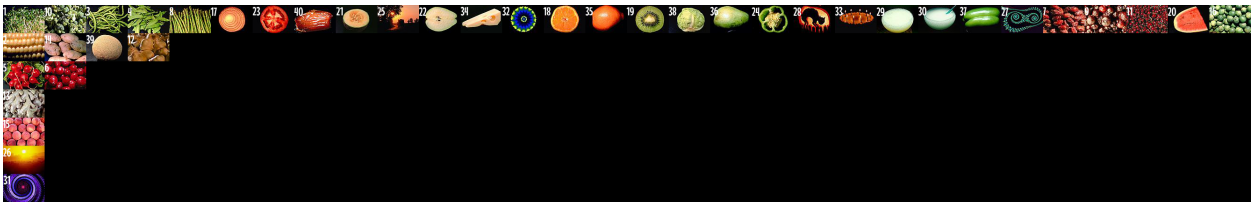
**Figure 8.** Clusters made using complete link, using r-values of 0.4 (top), 0.5 (middle), and 0.6 (bottom).

The last two differences made r = 0.6 a non-sensical choice for me, and I ultimately ended up going with r = 0.5. Even though r = 0.6 gave significantly higher average user recall, some of the clusters generated by my program made no sense and did not reflect something that most users would do. I decided to strike a balance between user recall scores and whether the clusters generated by my algorithm made sense and reflected a cluster that an actual human might make.

The single link clusters barely change between the different r-values, which makes sense considering the rationale behind them – each image in a cluster is very similar to one other image, but not necessarily similar to the other images. For single link, the single-image clusters are usually one of the four most different images for color and texture (ex. Images 15 and 31 were among of the four most different images for color, and Image 26 was one of the most different images for texture), which makes sense as these images don't fit in well with anything else in terms of either color or texture, and with a r-value of 0.4-0.6, both color and textural differences have a considerable impact on cluster formation.

**Figure 9.** Clusters made using single link, using r-values of 0.4 (top), 0.5 (middle), and 0.6 (bottom).

Melanie Hsu
mlh2197



**Snippet 14a.** Code for complete link clustering – initializing the dictionary of clusters.

```
#pre-set the clusters
        clusters = {}
        for i in range(0, 40):#initialize a 2D array to store the distances (1-linear sum)
           clusters[i] = []
           clusters[i].append(i+1)
```

To make the clusters, I first set up a dictionary of arrays. In each array entry, I added a number that corresponded to the number of one of the forty images. This resulted in a dictionary of forty arrays, representing the forty single-image clusters:

{0: [1], 1: [2], 2: [3], 3: [4], 4: [5], 5: [6], 6: [7], 7: [8], 8: [9], 9: [10], 10: [11], 11: [12], 12: [13], 13: [14], 14: [15], 15: [16], 16: [17], 17: [18], 18: [19], 19: [20], 20: [21], 21: [22], 22: [23], 23: [24], 24: [25], 25: [26], 26: [27], 27: [28], 28: [29], 29: [30], 30: [31], 31: [32], 32: [33], 33: [34], 34: [35], 35: [36], 36: [37], 37: [38], 38: [39], 39: [40]}

**Snippet 14b.** Code for complete link clustering – the first cluster fusion.

```
        #reduce to 39 clusters
        smallest = 1.0
        first = 0
        second = 0
        for i in range(0, 40):
           for j in range(0, 40):
                    if i == j:#don't compare an image to itself
                             continue
                    one = linsum[i+1]#first image to compare
                    two = linsum[j+1]#second image to compare
                    difference = one[j+1]
                    if difference < smallest:#logic to find the two closest images
                             first = i+1
                             second = j+1
                             smallest = difference
        clusters[first-1].append(second)#add the contents of second cluster to the first cluster
        clusters.pop(second-1)#remove the cluster that the second img belongs to
```

Next, I used two for loops so that I could compare the distances between every pair of images in the dataset, and find among them the pair of images with the smallest distance. At the end of the two for loops, the variables *first* and *second* would contain the keys corresponding to the image numbers of the

two nearest images in the dataset, while the variable *smallest* would hold the difference between these two images. Now that the two closest images have been found, I add the second image to the cluster that contains the first image, then delete the cluster containing the second image from clusters. At the end of this section, there are 39 clusters: 38 single-image clusters and one cluster with a pair of images.

The cluster in red was the newly formed cluster. In this case, Images 7 and 9 were judged to be the two closest images, so Image 9 was added to the cluster containing Image 7, and Image 9's former cluster was deleted from the dictionary.

{0: [1], 1: [2], 2: [3], 3: [4], 4: [5], 5: [6], 6: [7, 9], 7: [8], 9: [10], 10: [11], 11: [12], 12: [13], 13: [14], 14: [15], 15: [16], 16: [17], 17: [18], 18: [19], 19: [20], 20: [21], 21: [22], 22: [23], 23: [24], 24: [25], 25: [26], 26: [27], 27: [28], 28: [29], 29: [30], 30: [31], 31: [32], 32: [33], 33: [34], 34: [35], 35: [36], 36: [37], 37: [38], 38: [39], 39: [40]}

**Snippet 14c.** Code for complete link clustering – going down to 7 clusters.

```
#keep doing this until the clustering until number of clusters = 7
while len(clusters) > 7:
  mostfar = 1.0
  mostfar1 = []#will eventually store the
  mostfar2 = []#two clusters to be merged
  for keyA in clusters.keys():#compare each cluster to every other cluster
        for keyB in clusters.keys():
              #key no longer in dict cause of merging with another cluster
              if keyA not in clusters or keyB not in clusters:
                    continue
              if keyA == keyB:#don't compare a cluster to itself
                    continue
              gloop = clusters[keyA]
              ploop = clusters[keyB]
              farthest = 0.0
              far1 = 0
              far2 = 0
              for k in range(len(gloop)):#compare the farthest images in each cluster
                    for l in range(len(ploop)):
                          one = linsum[gloop[k]]
                          two = linsum[ploop[l]]
                          difference = one[ploop[l]]
                          if difference > farthest:
                                far1 = gloop[k]
                                far2 = ploop[l]
                                #this ends up as the distance measure
                                #between these two clusters
                                farthest = difference
                          if farthest < mostfar:#find the two nearest clusters (to merge)
                                mostfar = farthest
                                #these end up as the two clusters to merge
                                mostfar1 = keyA
                                mostfar2 = keyB
  A = clusters[mostfar1]
  B = clusters[mostfar2]
  for i in range(len(B)):
        A.append(B[i])#append contents of cluster B to cluster A
        clusters.pop(mostfar2)#remove cluster B from the dictionary
```

Melanie Hsu
mlh2197

This section of code runs until the number of clusters in the array is reduced to 7, signified when len(clusters) == 7. While the code for reducing 40 -> 39 clusters dealt purely with single-image clusters, this section of code must deal with both single and multiple-image clusters.

The two outer for loops (for keyA in clusters.keys(): and for keyB in clusters.keys():) shown are used to compare every cluster with every other cluster, while the two inner for loops (for k in range(len(gloop)): and for l in range(len(ploop)):) compare each member in the first cluster being compared with each member in the second cluster. The purpose of these loops is to first find the farthest pair of images in the two clusters being compared, then use the distance between this pair of images as the distance between these two clusters. The loop then finds the two clusters with the smallest distance.

Since nearness between two clusters is defined as the closest that the farthest image in one cluster comes to the farthest images in the second cluster, in order to find the distance between two clusters, I must first find the two images in a cluster that are farthest from each other.

After we've compared each member of the first cluster to every member of the second cluster, the variables *farthest*, *far1*, and *far2* will hold information about the two most farthest images between two clusters. After we've iterated though all of the keys in the dictionary, the variables *mostfar*, *mostfar1*, and *mostfar2* will store information about the two clusters that are most similar to each other.

In order to combine the two clusters, I set A = clusters[*mostfar1*] and B = clusters[*mostfar2*], where *mostfar1* and *mostfar2* are dictionary keys whose values are the arrays containing the two closest clusters. After adding each member of cluster B to cluster A, cluster B is then deleted from the dictionary, thus decreasing the cluster count by 1.

After the while loop is done, the dictionary contains exactly seven clusters:
{0: [1, 10, 3, 4, 8, 16, 5, 6, 7, 9, 11, 15], 1: [2, 12, 14, 39], 12: [13], 16: [17, 23, 40, 28, 33, 21, 25, 22, 19, 38, 24, 18, 35, 36, 20, 27, 32, 34], 25: [26], 28: [29, 30, 37], 30: [31]}

The code for calculating clusters by single link is almost identical, with the only difference being the reversal of the < sign in the following line:

```
if difference > farthest:
```

This is because single link defines the distance between two clusters as the closest that the nearest image in one cluster comes to the nearest image in the second cluster.

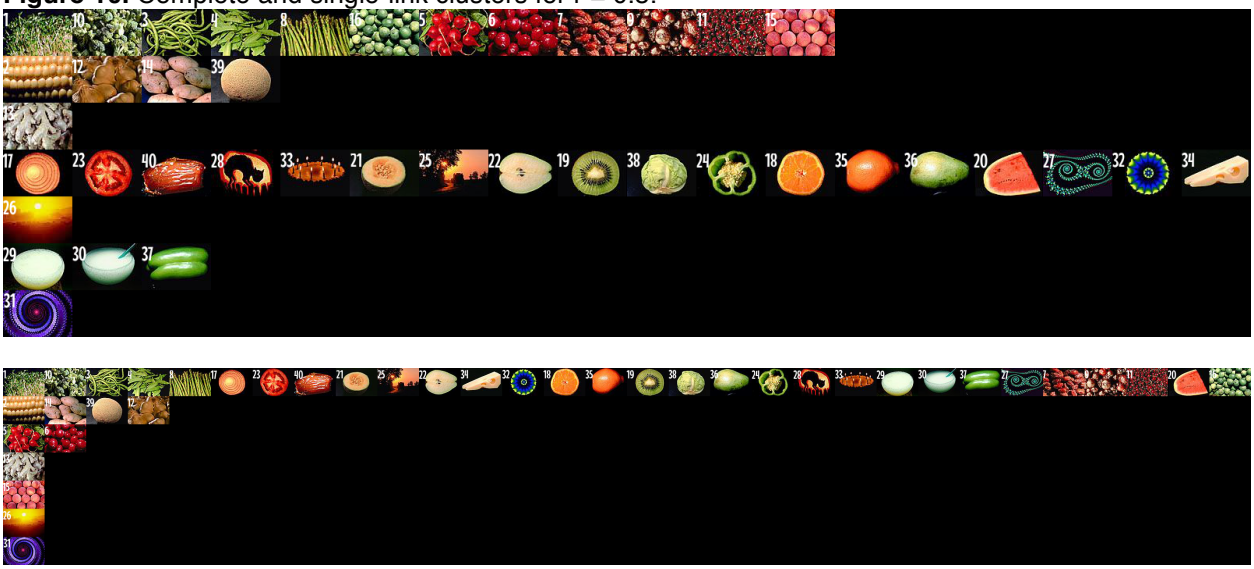
As I mentioned before, the single link clusters are very un-representative of how a person would cluster images. Of the seven single-link clusters, four are single-image clusters and a fifth one only contains two images. The longest cluster contains a large cluster of vastly different images, from the pale and texturally smooth liquids (Image 29 and 30), to the rough and vivid green vegetables (Images 1, 10, and 8). It's clear that each image in the set has at least one image that's similar to it in color and texture, but when viewing the cluster as a whole, it seems like a bunch of weakly-related images were thrown together.

Furthermore, Images 5 and 6 ended up in their own cluster, even though they are similar in both color and texture to Images 7, 8, and 11; yet, the completely different Images 29 and 30 found their way into the cluster containing those three images. The single link clusters are thus terrible because not only are unrelated images in the same cluster, but simliar images are also sometimes stuffed into different clusters. These observations hold true for single link clusters regardless of the r-value used.

In contrast, the complete link clusters is very good and consistently pairs similar images with each other. Images 31 and 13, which are different in color from everything else, and Image 26, which is extremely different texturally from most of the dataset, are understandably in their own clusters. For the cluster of

three, Images 29 and 30 are very closely related in color, with Image 37 and Image 30 being one of the top three color matches for each other, and all three are closely related in texture. However, it is unlikely that a user will place these three images together, since Image 37 is more likely to be classified with the other green vegetables due to its color. The cluster of four (Images 2, 12, 14, and 39) are fairly similar in color and texture, but not incredibly close to each other if you consider either pure color or pure texture.

**Figure 10.** Complete and single-link clusters for r = 0.5.



The second largest cluster, which consists of red and green fruits and vegetables, are closely related in either color (all of the green vegetables are each other's close matches, same with all of the red vegetables), or in texture (all are moderately to heavily textured). Finally, the largest cluster, which seems miscellaneous at a first glance, is filled with objects that both color or texturally similar – a lot of the images in the group are each other's top three matches.

**Figure 11.** The reasons behind the largest complete link cluster: for color (first two columns), each cluster member is almost always matched with other members of the cluster. For texture (last two columns), each cluster member is again almost always matched with other members of the cluster.

Melanie Hsu
mlh2197

Overall, the organization of the seven clusters is expected given that texture and color both have equal contributions when r = 0.5, and the algorithm overall made clusters whose members are very closely related to each other in either or both color and texture.

Even though the algorithm did a great job of grouping images together based on color and texture, the output does not strongly reflect how a real person would cluster these images. Due to the numerous factors that go into image categorization, it's very hard if not impossible to algorithmically replicate exactly why people place certain images in certain clusters. A person would likely categorize pictures based on their content, rather than just color and texture. Most likely, they would place the fractals into one group, place the two liquids into another group, and classify the rest of the images based on either color (red foods vs. green foods), shape, or state (cut foods vs. whole foods). The two users who justified their cluster choices indeed categorized the images primarily based on what the images represented. I will discuss this more about this algorithm's inability to predict actual user choices in Part 4.

**Part 4: User Study**
The python file for this part is **user.py**.

For the user study, I asked three different people to perform three tasks:

1) Color task: For each of the forty images, find the image that's most similar to it and the image that's most different from it for color. The users' choices for this task are shown in Table 2.
2) Texture task: For each of the forty images, find the image that's most similar to it and the image that's most different from it for texture. The users' choices are displayed in Table 3.
3) Clustering: Group the forty images into seven clusters. The clusters made by each user, and their rationales (if provided), are shown in Table 4.

More information about the users:
User 1 is an artist and animator, User 2 is a cook, and User 3 is a programmer.

**Table 2.** Images assessed by most and least similar color, according to the three users

| Image | Most similar – User 1 | Most similar – User 2 | Most similar – User 3 | Least similar – User 1 | Least similar – User 2 | Least similar – User 3 |
|---|---|---|---|---|---|---|
| 1 | 10 | 10 | 10 | 23 | 33 | 31 |
| 2 | 6 | 34 | 39 | 31 | 37 | 31 |
| 3 | 4 | 4 | 16 | 18 | 13 | 31 |
| 4 | 3 | 36 | 8 | 5 | 14 | 31 |
| 5 | 6 | 23 | 6 | 4 | 32 | 13 |
| 6 | 5 | 7 | 11 | 4 | 31 | 13 |
| 7 | 11 | 40 | 40 | 10 | 36 | 13 |
| 8 | 3 | 16 | 4 | 15 | 26 | 31 |
| 9 | 6 | 11 | 40 | 10 | 27 | 32 |
| 10 | 16 | 8 | 16 | 15 | 20 | 31 |
| 11 | 7 | 6 | 6 | 10 | 29 | 29 |
| 12 | 14 | 14 | 14 | 31 | 36 | 31 |
| 13 | 39 | 39 | 39 | 7 | 6 | 31 |
| 14 | 12 | 21 | 12 | 31 | 5 | 31 |
| 15 | 6 | 35 | 11 | 4 | 8 | 16 |
| 16 | 3 | 19 | 3 | 5 | 35 | 31 |
| 17 | 18 | 18 | 25 | 39 | 16 | 31 |
| 18 | 17 | 35 | 25 | 39 | 23 | 31 |
| 19 | 3 | 36 | 36 | 20 | 6 | 31 |
| 20 | 15 | 23 | 15 | 19 | 2 | 31 |
| 21 | 34 | 17 | 25 | 19 | 37 | 31 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 22 | 21 | 36 | 19 | 31 | 4 | 31 |
| 23 | 35 | 40 | 5 | 10 | 3 | 32 |
| 24 | 16 | 19 | 36 | 23 | 7 | 31 |
| 25 | 26 | 26 | 17 | 4 | 19 | 37 |
| 26 | 25 | 25 | 25 | 4 | 29 | 37 |
| 27 | 8 | 32 | 16 | 23 | 15 | 6 |
| 28 | 23 | 33 | 17 | 31 | 37 | 31 |
| 29 | 30 | 30 | 36 | 23 | 9 | 31 |
| 30 | 29 | 29 | 29 | 23 | 11 | 31 |
| 31 | 32 | 32 | 32 | 20 | 12 | 13 |
| 32 | 31 | 27 | 3 | 20 | 26 | 39 |
| 33 | 28 | 40 | 25 | 31 | 16 | 16 |
| 34 | 21 | 17 | 2 | 30 | 20 | 31 |
| 35 | 18 | 18 | 18 | 36 | 36 | 19 |
| 36 | 29 | 19 | 16 | 35 | 40 | 35 |
| 37 | 19 | 16 | 27 | 17 | 18 | 31 |
| 38 | 16 | 36 | 36 | 18 | 17 | 31 |
| 39 | 34 | 21 | 14 | 2 | 7 | 31 |
| 40 | 35 | 7 | 33 | 29 | 4 | 16 |

**Table 3.** Images assessed by most and least similar texture, according to the three users

| Image | Most similar – User 1 | Most similar – User 2 | Most similar – User 3 | Least similar – User 1 | Least similar – User 2 | Least similar – User 3 |
|---|---|---|---|---|---|---|
| 1 | 10 | 3 | 10 | 30 | 37 | 37 |
| 2 | 16 | 7 | 15 | 26 | 36 | 33 |
| 3 | 4 | 8 | 8 | 26 | 39 | 30 |
| 4 | 3 | 1 | 3 | 26 | 35 | 30 |
| 5 | 6 | 6 | 4 | 26 | 36 | 26 |
| 6 | 5 | 9 | 11 | 26 | 38 | 26 |
| 7 | 9 | 11 | 40 | 26 | 40 | 30 |
| 8 | 3 | 3 | 3 | 26 | 29 | 29 |
| 9 | 6 | 5 | 7 | 26 | 30 | 29 |
| 10 | 9 | 16 | 13 | 26 | 32 | 29 |
| 11 | 5 | 6 | 6 | 26 | 21 | 29 |
| 12 | 14 | 10 | 5 | 26 | 31 | 29 |
| 13 | 14 | 10 | 12 | 26 | 20 | 29 |
| 14 | 12 | 12 | 36 | 26 | 27 | 29 |
| 15 | 5 | 16 | 16 | 26 | 37 | 29 |
| 16 | 9 | 6 | 6 | 26 | 17 | 29 |
| 17 | 18 | 32 | 31 | 26 | 26 | 26 |
| 18 | 17 | 22 | 19 | 26 | 38 | 26 |
| 19 | 16 | 18 | 18 | 26 | 3 | 26 |
| 20 | 15 | 21 | 22 | 26 | 4 | 26 |
| 21 | 34 | 19 | 22 | 26 | 1 | 26 |
| 22 | 21 | 19 | 21 | 26 | 5 | 26 |
| 23 | 24 | 20 | 24 | 26 | 13 | 26 |
| 24 | 23 | 28 | 23 | 26 | 14 | 26 |
| 25 | 26 | 26 | 26 | 1 | 10 | 29 |
| 26 | 25 | 25 | 25 | 1 | 30 | 29 |
| 27 | 31 | 31 | 31 | 26 | 12 | 26 |
| 28 | 33 | 33 | 32 | 26 | 8 | 29 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 29 | 30 | 30 | 30 | 26 | 11 | 26 |
| 30 | 29 | 29 | 29 | 26 | 5 | 26 |
| 31 | 32 | 32 | 27 | 26 | 16 | 26 |
| 32 | 31 | 27 | 27 | 26 | 14 | 26 |
| 33 | 21 | 25 | 30 | 26 | 22 | 26 |
| 34 | 21 | 33 | 4 | 26 | 3 | 29 |
| 35 | 17 | 36 | 36 | 26 | 27 | 26 |
| 36 | 18 | 39 | 35 | 26 | 31 | 26 |
| 37 | 19 | 35 | 40 | 26 | 8 | 26 |
| 38 | 16 | 16 | 40 | 26 | 25 | 26 |
| 39 | 34 | 21 | 35 | 26 | 26 | 26 |
| 40 | 35 | 7 | 7 | 26 | 2 | 26 |

Before comparing outputs, I rearranged the color and texture arrays that I obtained from Parts 1 and 2 in order of descending similarity, and placed the reordered array in a new dictionary of arrays. In the array for a given image, the first entry in the array holds the number of its most simlar image, and the last entry in the array holds the number of its most different image. Below is the array for Image 1:

1: [10, 3, 16, 19, 4, 8, 38, 36, 24, 29, 39, 22, 5, 37, 21, 32, 34, 2, 27, 30, 14, 40, 13, 25, 20, 9, 28, 33, 18, 35, 23, 7, 17, 31, 11, 12, 6, 26, 15]

This ordering allows me to assess how close my color and texture rankings are with the color and texture rankings of the three users. For example, for Image 1, all three users chose Image 10 as the most similar image in terms of color. Because my algorithm also ranked Image 10 as Image 1's most similar image, the distance between my ranking and their's is 0. For Image 2, however, my algorithm chose Image 39 as its most similar image. User 1 chose Image 6, User 2 chose Image 34, and User 3 chose Image 39.

Since User 3 and I both agree that Image 39 is the most simliar image for Image 2, the distance between our rankings is again 0. For Users 2 and 3, I determine the distance between our rankings by checking their choice of most similar image against each successive image in my array for Image 2, until there is a match between their choice and mine.

User 3's choice vs. algorithmic output for Image 2
39 vs. 39 – match; since their top choice is the same as the algorithm's, the difference in rankings is 0

User 2's choice vs. algorithmic output for Image 2
34 vs. 39 – mismatch, look at the second most similar image in the array
34 vs. 21 – mismatch, look at the third most similar image in the array
34 vs. 34 – match, their first choice is the algorithm's fourth choice

Their top choice turns out to an image that my algorithm ranked as the fourth most similar image to Image 2. Hence, the distance between our rankings is 4-1 = 3.

User 1's choice vs. algorithmic output for Image 2
6 vs. 39
6 vs. 21
…
6 vs. 13
6 vs. 6 – match!

2: [39, 21, 25, 34, 9, 17, 12, 28, 40, 22, 18, 8, 1, 14, 20, 35, 24, 33, 19, 5, 38, 23, 10, 36, 29, 11, 16, 3, 4, 27, 32, 30, 7, 37, 13, 6, 26, 15, 31]

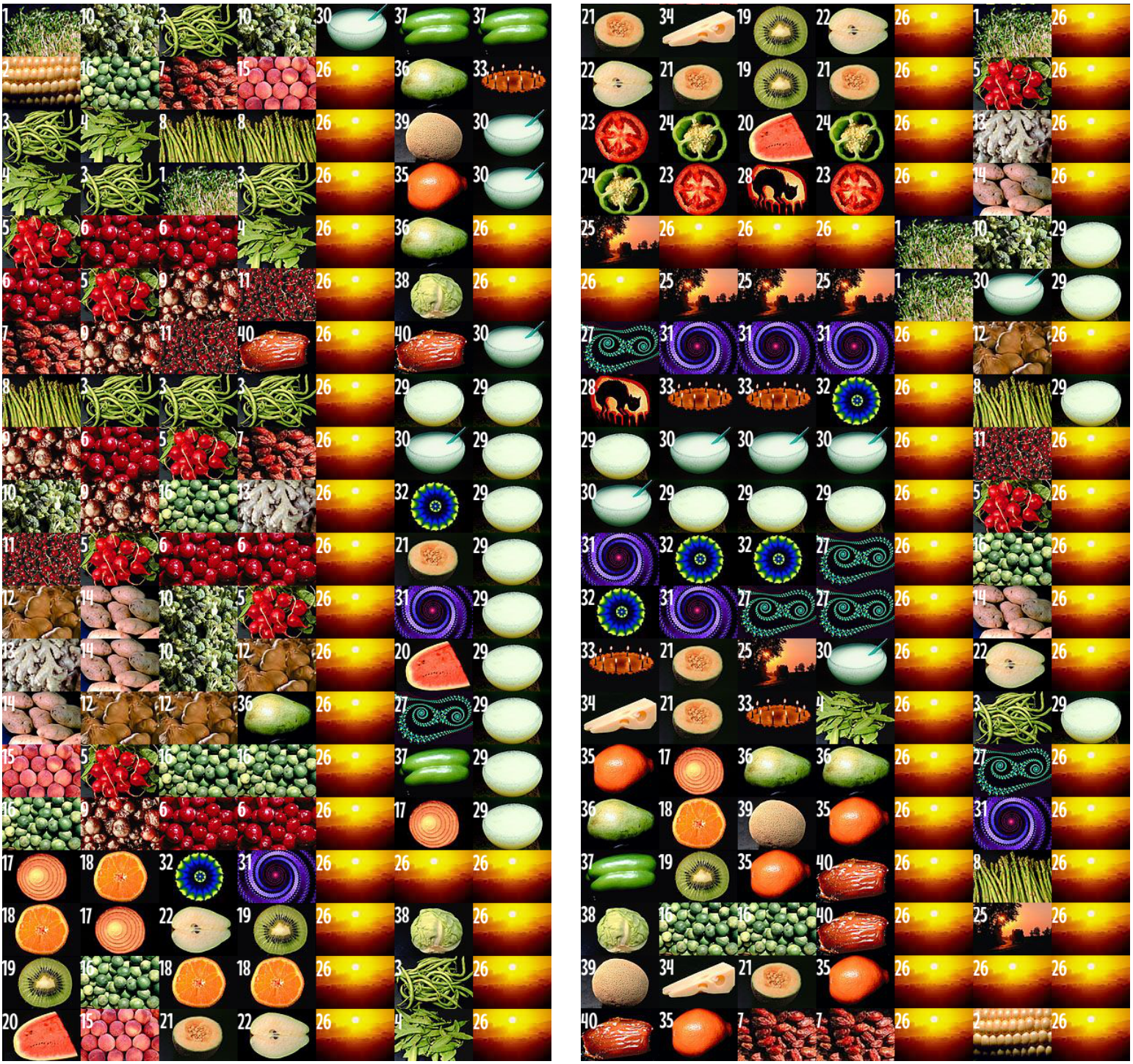
Melanie Hsu
mlh2197

**Figure 12.** User results for color task. Image was split into two for readability.
For each row:

Column 1: The image
Column 2: The image that is most simlar to Col. 1, according to User 1
Column 3: The image that is most similar to Col. 1, according to User 2
Column 4: The image that is most similar to Col. 1, according to User 3
Column 5: The image that is most different from Col. 1, according to User 1
Column 6: The image that is most different from Col. 1, according to User 2
Column 7: The image that is most different from Col. 1, according to User 3

Melanie Hsu
mlh2197

**Figure 13.** User results for texture task. Image was split into two for readability. The rows and columns are organized the same way as Figure 11.



User 1 chose an image that's extremely low on the similarity rankings for Image 2 – the 36th image, Image 6. The ranking distance is thus 36-1 = 35. In this case, User 1's prediction was extremely poor – Image 6 is the pretty much the opposite color of Image 2.

Melanie Hsu
mlh2197

**My initial approach:**
At first, I decided that using unadjusted distances to compare the algorithmic and user outputs was not enough: larger rank distances should have a large impact on the ranking similarity scores. I decided to use a variation of the Spearman rank correlation coefficient formula:

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)}.$$

I made a few changes to the formula, since my ranking situation did not fit the intended use of the formula: rather than comparing two fully ranked lists (ex. two people's rankings of ten colleges), I was ranking the top choice of one list against potentially all of the ranks in a second list. Using the unaltered formula in this situation would result in a misleading result. Instead, I used the following formula:

$$\text{Similarity} = 1 - \frac{\sum d_i^2}{n(n-1)^2}$$

Where n = number of images, and $\sum d_i^2$ means the sum of the squares of the ranking difference between the user and the algorithm outputs for every image in the set. I intentionally tweaked the numerator and denominator so that if the user and the algorithm had 100% agreement on all of the images (ex. the user and algorithm chose the same most similar image for every image in the dataset), then the similarity between the algorithmic and user outputs would be 1.0. Conversely, if the user chose the most different image as the most similar image for every image in the set (ex. they chose Image 15 as the most similar image for Image 1, Image 31 as the most similar image for Image 2, …), then the ranking difference for every single image would be 39, resulting in a similarity of 0.0 ($1 - 40*39^2/40(39)^2$).

Alt 3 is the implementation of the algorithm I described. The same algorithm is used for both color and texture comparsions between the algorithm and each user. To compare the users' most different image rankings against the algorithm's most differente image rankings, I iterate through the array backwards (starting at the 39th most similar image for the given image and ending at the most similar image for the given image) instead of forwards.

**Alt 3.** Calculating the distance between the algorithm's and User 1's color similarity rankings

```
csimdist1 = 0
#calculate the difference between ranks using a variant of the Spearman correlation
#I adjusted the formula so that if the user and I agreed on all rankings, the similarity would be 1.0,
#and if we disagreed completely on all rankings, the similarity would be 0.
total = float(40*(39**2))
i = 0
while i < 40:#for each image
  #for similarities, do comparisons from front of array to back
  colors = colorranks[i+1] #colorranks is the array that holds the alg's rankings for each image
  distance = 0#how many ranks apart is the user's choice of most similar image and
  #my choices of most similar image?
  for j in range(len(colors)):
      if int(colors[j]) != int(colorsim1[i]):
          distance += 1 #mismatch, increase the ranking distance by 1
          #see if the user's top choice matches the image number of the
          #next most similar image
          continue
      else:
          #the larger the distance between ranks, the more
          #it will affect the total distance measure
          csimdist1 += distance**2
```

Melanie Hsu
mlh2197

```
            break #there's a match, stop iterating through the rank array
        i += 1
    user1colorsims = 1 - (csimdist1)/total
```

**Table Alta.** The $\sum d_i^2$ between the user and algorithmic color and texture calculations. To get the similarity values, I divided each of the distances by the denominator of the equation, $40*39^2$.

| User | Color Similarity Rank Distance | Color Difference Rank Distance | Texture Similarity Rank Distance | Texture Difference Rank Distance |
|---|---|---|---|---|
| 1 | 8470 | 13979 | 8812 | 484 |
| 2 | 4437 | 8135 | 9124 | 13549 |
| 3 | 4661 | 4009 | 7433 | 4482 |
| Avg. | 5856 | 8707.666667 | 8456.333333 | 6171.666667 |

With a maximum $\sum d_i^2$ value of 60840 in the worst case, and a maximum $\sum d_i^2$ value of 0 in the best case, it's fair to say that a $\sum d_i^2$ value of 30420, which results in a similarity of 0.5, is considered an extremely bad user output and is no better than randomly picking most similar and most different color and texture images for every image. Since none of the users come close to a 0.5 similarity with the algorithm's output, then their outputs seem to be much better than a random image picker.

**Table Altb.** Comparison of user and algorithmic color and texture calculations. Similarity value of 1 means that the user achieved complete similarity with the program's output, 0 means that the user's output is completely different from the program's output.

| User | Color Similarity | Color Difference | Texture Similarity | Texture Difference |
|---|---|---|---|---|
| 1 | 0.860782380013 | 0.77023339908 | 0.855161078238 | 0.992044707429 |
| 2 | 0.927071005917 | 0.866288625904 | 0.85003287311 | 0.777301117686 |
| 3 | 0.92338921762 | 0.934105851414 | 0.877827087442 | 0.926331360947 |
| Avg. | 0.903747535 | 0.856875959 | 0.861007013 | 0.898559062 |

Because my algorithm penalizes harshly against repeated, large mistakes, but minimizes the effects of repeated small to medium mistakes (there are a lot of similarly-colored images in the database after all, so it's quite possible that the tenth most simliar image for a green image is still a similar shade of green), getting a similarity of over 0.90 was not considered impressive; if a user were 10 ranks off on every single image, they would still get a similarity score of 0.9342537804; being 8 ranks off for every single image would given them a score of 0.95792241946. However, across all three users and two tasks, there was only one score above 0.95, showing that high levels of accuracy are hard to achieve in practice.

**Table Altc.** Color task: How off each user was for each image, in terms of un-squared rank distance from the algorithm's choice of most simliar image for each image. Entries in red are considered egregious mistakes (worse than a random image picker).
1: [0 35 0 0 0 1 1 0 2 1 1 2 6 12 3 2 6 3 2 33 21 2 23 25 32 9 21 4 0 0 0 26 0 20 0 7 32 24 0 4]

2: [0 3 0 1 1 2 2 25 1 5 2 2 6 13 11 10 6 0 21 18 0 23 0 1 32 9 0 0 0 0 0 0 3 6 0 20 20 0 4 6]

3: [0 0 13 3 0 0 2 3 3 1 2 2 6 12 2 2 0 1 21 33 3 30 13 0 0 9 24 3 1 0 0 13 5 18 0 21 4 0 3 2]

User 1 makes many large mistakes, on eleven occasions selecting images that are past the middle of the array for the most similar image. Based on these observations, and the fact that there aren't this many images in th database with similar colors, I considered user 1's similarity score of 0.860782380013 to be bad, as they were picking the image with very different colors as the best matches for a lot of the images.

Users 2 and 3 made fewer mistakes, but even so, they only achieved scores of 0.92 due to User 2 making quite a few moderate mistakes, and User 3 making a medium mistake and two large mistakes.

Since their accuracies for most images were very high (< 10 ranks off), for this data set, I consider a score of 0.92 or higher to be fair. A score of over 0.98 (average 5 ranks off per image) is good.

In short:
< 0.74 (average 20 ranks off) = really bad
< 0.85 (average 15 ranks off) = bad
> 0.93 (average 10 ranks off) = fair
> 0.98 (average 5 ranks off) = good

Ultimately, I decided that this was a pretty bad algorithm, as correlations above 0.85 shouldn't be considered "bad," and the system minimized smaller mistakes (< 15 ranks off) so much that it took a lot of egregious errors to get the similarity to fall below 0.9. In the database, the most prevalent dominant color is green (11 images), followed by red or pink (5 images), and orangeish (4 images) – and this is including only the images with undisputable dominant colors. That means that for most images, being more than 5 ranks off is considered *very* inaccurate.

**My revised approach:**
Based on these results, I decided to revise my algorithm to give equal weight to all rank differences. The the core of the algorithm was the same as that of the alternate implementation – compare the user's choice of most similar (or least similar when comparing the most different images) image against each successive member of the given image's array until we have a match. Then, add the rank distance between the user's choice and the algorithm's choice to a *distance* variable. By the end, the *distance* variable will store the total rank distance across 40 images. I then divide this total rank distance by 40 to obtain the average rank distance per image for the user.

**Snippet 15.** Calculating the distances between the algorithm's and User 1's color similarity rankings. The same algorithm is used for both the color and texture tasks.

```
#start with a distance of 0, and add 1 for the distance of each rank
        #at the end, divide by 40 to get the average rank distance for each user
        total = float(40)
        i = 0
        while i < 40:#for each image
          #for similarities, compare from front of array to back
          colors = colorranks[i+1]
          distance = 0#how many ranks apart is the user's most similar and my most similar image?
          for j in range(len(colors)):
                  if int(colors[j]) != int(colorsim1[i]):
                          distance += 1
                          continue
                  else:
                          print distance
                          csimdist1 += distance
                          break #match!
          i += 1
```

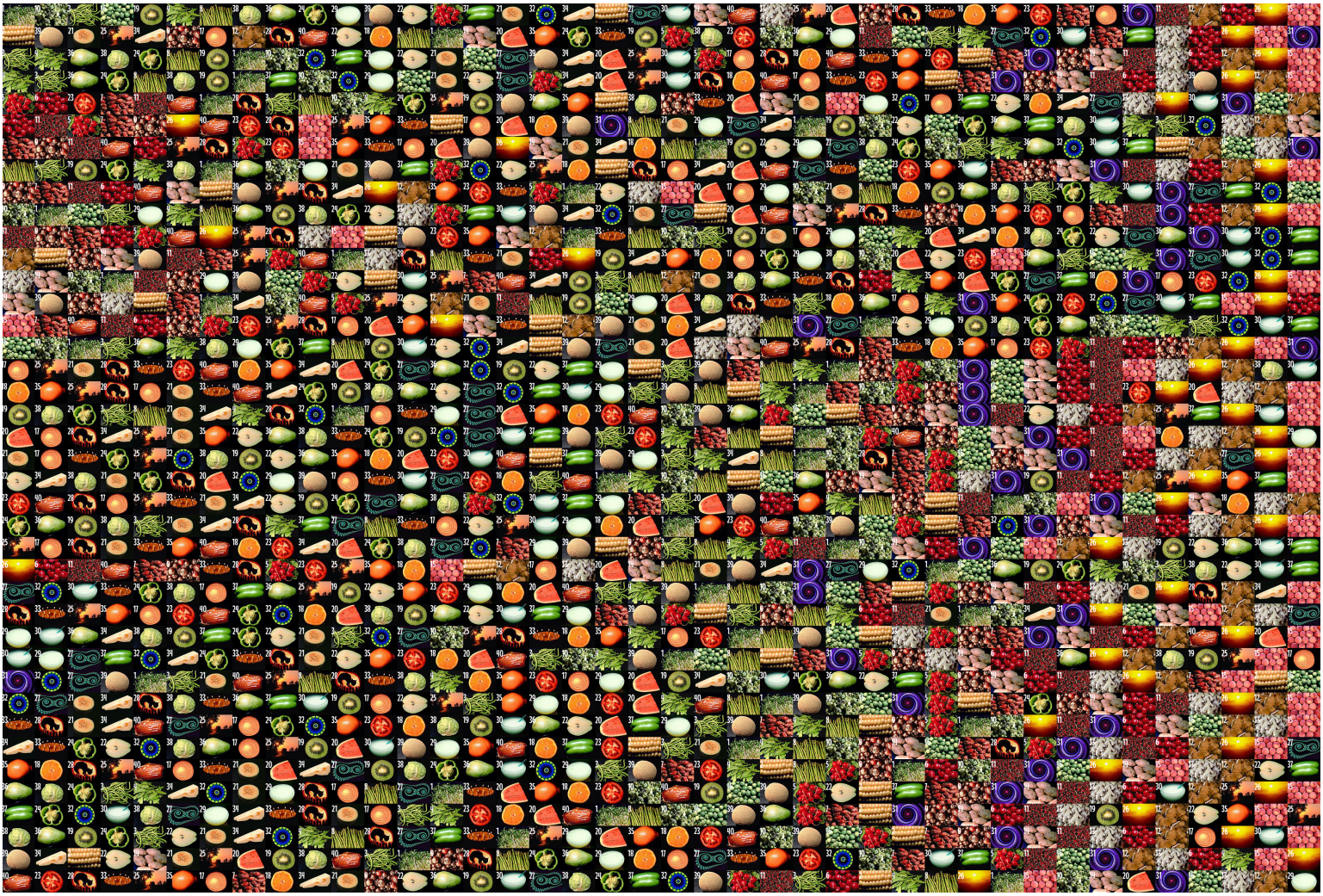
**Table 4.** Average rank differences for the three users on the color and texture task.

| User | Color Similarity | Color Difference | Texture Similarity | Texture Difference |
|---|---|---|---|---|
| 1 | 9 | 15.875 | 10.75 | 1.35 |
| 2 | 6.325 | 11.925 | 11.7 | 14.975 |
| 3 | 6.375 | 7.825 | 10.275 | 5.5 |
| Avg. | 7.233333333 | 11.875 | 10.90833333 | 7.275 |

Melanie Hsu
mlh2197

To better visualize the rankings for each image and how accurate each user was, I built Figure 14 for the color image rankings.

**Figure 14.** 40x40 table of all image rankings by color. Each row depicts an image ranking array for each image in the dataset. The first column in each row contains the image. The remaining columns in each row contain the other 39 images, ordered from left to right based on how similar they are in color to the image in the first column (left: most similar, right: most different).



For color simliarities, the images tend to start looking pretty different in color after the fifth most simliar image (on average); because the most different images can be very different from each other in terms of color, it's harder to say when these images start to look too similar to the given image. Top five different images seems like a safe bet, as the majority of the five most different images for each image fall in the opposite or at least a very different section of the color wheel.

Based on these assumptions, Users 2 and 3 reached significant agreement with the algorithm for most similar color selections, having average rank differences of 6.325 and 6.372 respectively. Due to the high number of egregious mistakes they made, User 1 had relatively poor agreement, with an average rank difference of 9 – the 9$^{th}$ most simliar image for many images has already hopped over to another section of the color wheel, and can't be considered the same color (ex. turquoise to reddish-orange for Image 27).

As for most different color selections, Users 1 and 2 did not do so well, having rank difference averages of 15.875 and 11.925 respectively. User 3 did much better, with a rank difference average of 7.825. Often times, an image's 15th most different image is sometimes already in the adjacent section of the color wheel, and can be considered more similar color-wise to the image than different; for instance Image 3's 15th most different image is orange in color. However, if we look at only the ten most different images of a given image, most of them are sufficiently different from the given image to be clasified as having a "very different" or "opposite" color. Since the consequences of a higher rank difference is lessened when it comes to selecting the most differently-colored image, I think I can safely say that User 3 did a great job of selecting most different images for the color task, and User 2 actually did a decent job.

Since it was likely that User 1 was very distracted during the color task, judging from the high number of serious ranking mistakes they made, I treated their results as outlying for this task. The other two users, on average, had high agreement with the algorithm for color similarity, and moderate agreement for color difference. Since many images (especially the green vegetables) were very similar in color, and a lot of other images had no obvious "top friend" or "top foe." For instance, Image 13, which is whitish, and Image 31, which is purple, are very hard to rank since there are no other images in the set with the same or similar colors. Hence, it's expected that people will struggle to find the most similar and different image for quite a few of these images.

While the users and system agree well on color similarities, this high level of agreement isn't necesssarily an indication that the system is good overall. For one, the majority of images in this small database can be grouped into similarly colored foods and drinks. Because most of the images are food or drinks, the users almost always selected a similarly-colored food or drink as the most similar image. However, once an outlier appears (ex. Images 13, which is the only white image in the database, and Image 31, which is not only the only purple image but also a fractal), the users and algorithm often disagree on which image is most similar to it. While the algorithm picks the most similar image based on color alone, the user uses both color and content. This is evidenced when the algorithm picked Image 17 as the closest match to Image 26, yet all three users chose Image 25 – another image of a sunset – as the top match. In the system, Image 25 is only the tenth most simliar image to Image 26. In real life, the users will probably want the system to return other images of orange sunsets.

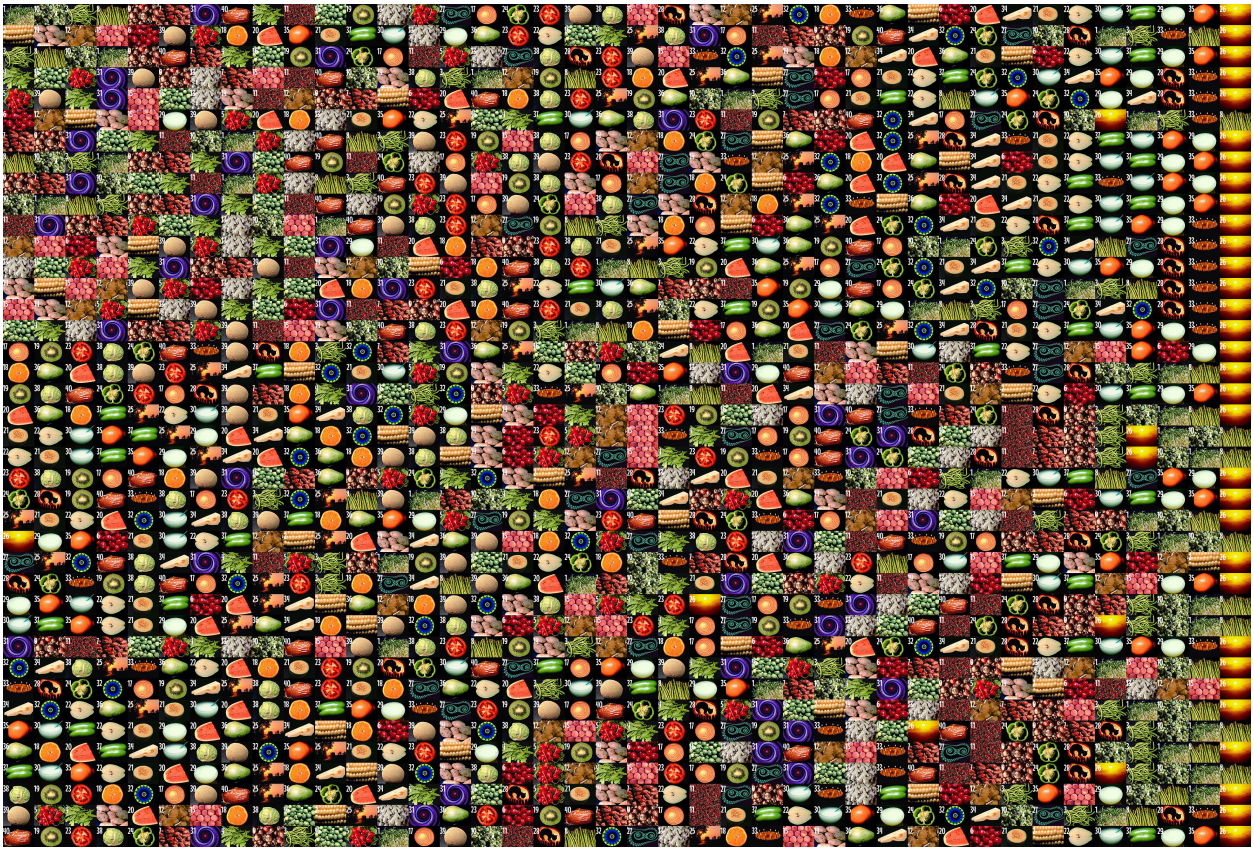
Indeed, one of the flaws of the system is that it either over-simplifies or doesn't take into account certain factors motivating the users' most different image rankings, such as the content of the image. For example, User 3 listed Image 31 as the "top foe" for over half of the images (the green, orange, and yellow foods). A system like this doesn't offer what users are looking for. Two colors don't have to be near-opposites for them to be considered very different by a person. Instead of ranking images by how different they are from the given image, and returning only the images that the algorithm considers most different, a more useful system should return several clusters of images, grouped together by both color and content. For instance, if the user is looking for something like "food opposite color of grapes" the system should probably return orange fruits and green vegetables, instead of images with vastly different subjects, such as orange fractals and orange cars.

In short, the disagreements between the users and system, mainly in the color difference task but also during some instances in the color similarity task, suggests that the system does not consistently meet the needs and expectations of its users.

For texture, I made a similar 40x40 image depicting all rankings for the forty images.

**Figure 15.** 40x40 table of all image rankings by texture. Each row depicts an image ranking array for each image in the dataset. The first column in each row contains the image. The remaining columns in each row contain the other 39 images, ordered from left to right based on how similar they are in texture to the image in the first column (left: most similar, right: most different).

Melanie Hsu
mlh2197



Looking at the texture table, it seems that the top 6-7 most similar images for a given image tend to have very similar or similar textures, while the top 6-7 most different images tend to be very texturally different from the given image. Images that fall into the middle texture class (moderately textured) have a lot of images that can be considered different from them, since both very rough and smooth objects are considered to be their “foes.” Also, since the dataset has a large proportion of both roughly and moderately textured objects, some images can have up to 10-12 very similar or very different image matches. For this reason, I defined a rank difference average of around 7 as strong system-user agreement on the texture task.

For texture simlarity, all three users had rank difference averages of over 10 (10.75 for User 1, 11.7 for User 2, and 10.275 for User 3), signifying considerable disagreements between the users and the system. While the system calculates and compares textures using local changes, people tend to use the overall “feel” of the image. Because most images in the database are surrounded by black backgronuds and/or have many shadows, the system has a tendency to over-estimate the amount of texture in an image. Since users don’t consider the black background when comparing the textures of different images, this over-estimation can lead to greater disagreements between the user and system.

This over-estimation becomes a problem for some images, such as Image 5. Although all three users picked Image 6 as the top match for Image 5, the algorithm ranked Image 6 as only the 13$^{th}$ most similar image. While all three users felt that Images 5 and 6 felt similar, because Image 6 was less intense overall and had more gradual local changes in color, it was considered by the system as much smoother than Image 5. This disagreement further supports a mismatch between the factors underlying the system and user texture ranks, and suggests that the system does not match user expectations and needs.

Two of the three users did exceptionally well on the texture differences task – User 1 had an average rank difference of only 1.35, meaning they agreed with the algorithm’s top match most of the time, and User 3

had an average rank difference of 5.5. However, the high levels of agreement may be a false positive, and changing the images in the dataset may cause this effect to disappear.

For texture differences, every image had either Image 26 or Image 8 as the "top foe." Because two of the three users felt that the texture of air and liquid was different from the texture of everything else, they often chose either Image 26 or Image 29 as the most different image. There are cases when considerations of the material being depicted in the image cause significant disagreements between the system and user. For instance, both Users 1 and 3 listed Image 26 as the "top foe" for Images 29 and 30, based on the fact that water can be "felt" by the hand, but not air, and air has significantly less resistance compared to water. Image 26 is the 5$^{th}$ most different image for Image 30, but only the 18$^{th}$ most different image for Image 29, due to a difference in lighting between the two liquid images.

Users don't care nearly as much about the shadows as the system does (unless the shadows help them decide how textured the object in the image is), and subtracting the first 32 black pixels isn't enough: even the lighter, grayish shadows in an image can affect the system's interpretaton of how textured the image is. Thus, in order to achieve better system-user agreement, the system has to treat the shadows in each image differently, based on the texture of the object(s) in the image. For instance, hard shadows on a smoother object should probably be ignored, otherwise the system will over-estimate the roughness of the objects in the image. In contrast, images of rougher objects will probably have a lot of hard shadows because of their texture, and therefore the shadows should go into the system's texture calculations.

Because of the different way in which the users and system define texture, the high level of agreement in the texture difference task is misleading. Overall, the users and system have a reasonable level of agreement on the texture task, but that's likely because most of the similarly-textured images have similar amounts of intensity and shadow.

**Table 5:** The images divided into seven clusters, according to three users. If the user provided a reason for making the clusters, the reason was listed with the cluster.

User 1:

1) 2 17 18 35
2) 24 36 19 38 16 37 10 4 3 8 1
3) 27 31 32
4) 28 33 26 25
5) 14 12 39 21 34 22
6) 13 29 30
7) 5 6 11 9 40 23 7 20 15

User 2:

1) 35 36 37 38 39 40
2) 1 3 4 8 10 13
3) 2 5 6 7 9 11 12 14 15 16
4) 29 30
5) 27 31 32
6) 17 18 19 20 21 22 23 24
7) 26 25 28 33 34

User 2's rationale:

Group 1: whole objects
Group 2: stringy objects
Group 3: beady objects
Group 4: liquid objects
Group 5: fractal patterns
Group 6: cut objects
Group 7: miscellaneous

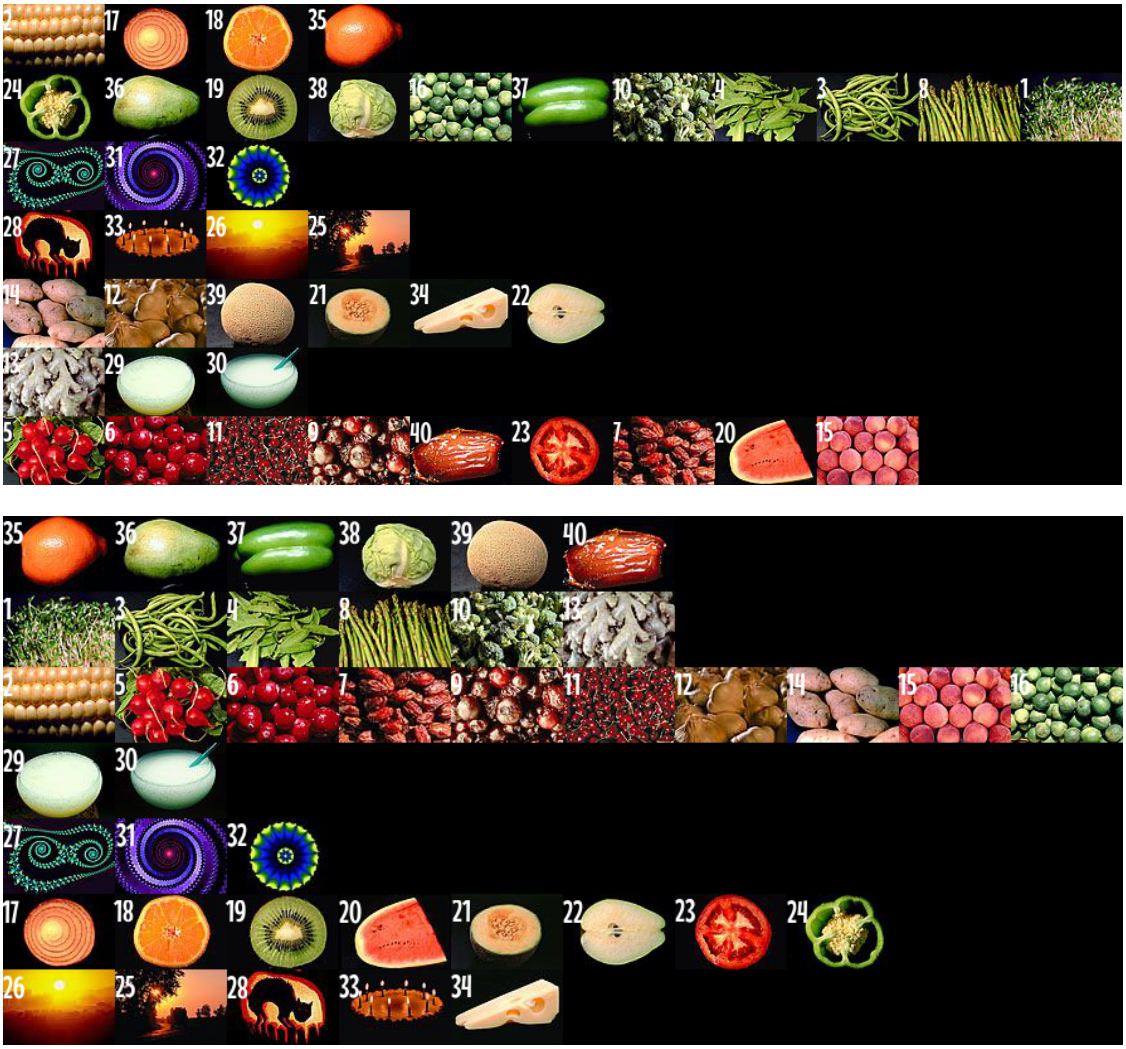
User 3:

1) 17 18 19 20 21 22 23 24
2) 12 13
3) 33 34
4) 5 6 7 9 11 15 35 36 39 40
5) 1 2 3 4 8 10 14 16 37 38
6) 29 30
7) 25 26 27 28 31 32

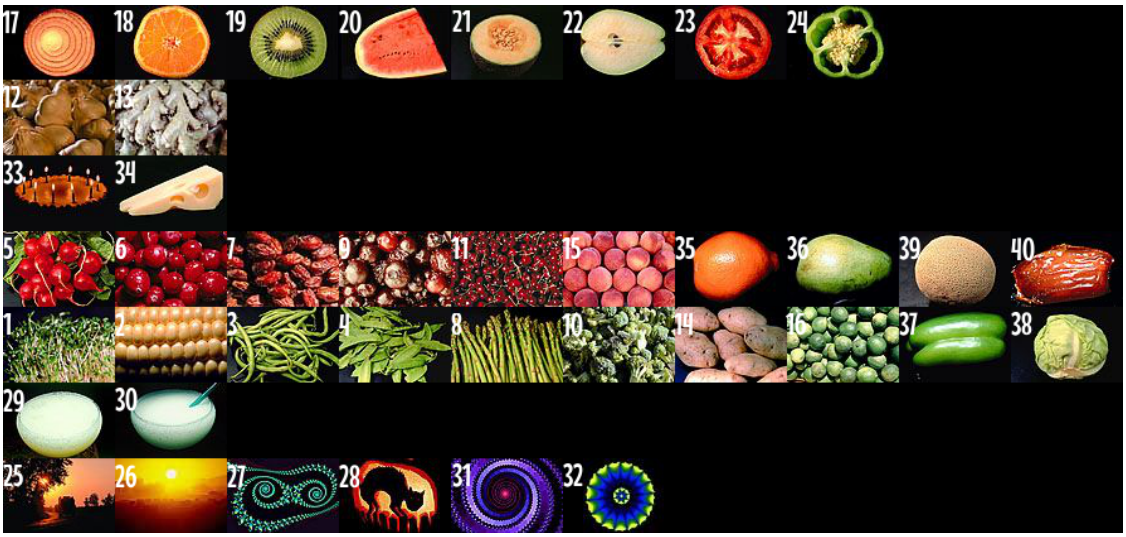
User 3's rationale:

Group 1: cut fruits and vegetables
Group 2: herbs and spices
Group 3: dairy and carbohydrates
Group 4: fruits
Group 5: vegetables
Group 6: food in bowls
Group 7: stuff you can't eat

**Figure 14.** The seven clusters made by user 1 (top), 2 (middle) and 3 (bottom).



Melanie Hsu
mlh2197



There is clearly going to be a tremendous disagreement between each user's clusters and the system's clusters, since two of the three users stated content-related reasons for the organization of their clusters (ex. object shape, how the object is used in real life, the state of the object). The user who did not provide a rationale, User 1, seemed to organize images by roughly by color, using a different color definition compared to the system. For instance, User 1 grouped the red and reddish-brown images into one cluster, whereas some of the red and reddish-brown images are ranked as very dissimilar by the system. Regardless of which measure I use, I predict a very poor user-system agreement for all three users.

I use precision and recall as the measures for comparing the user's and system's clusters. For each of the forty images, I compare the cluster that the user put the image in against the cluster that the system put the image in. If the two clusters are similar (they have many other images in common), then the recall and precision will both be high, as this signifies that the user and system used the same or similar rationale for defining their clusters. If the two clusters are very different, then the recall and precision will both be low. I defined recall as the number of images shared by the system and user cluster/number of images in the system's cluster, and precision as the number of images shared by the system and user cluster/number of images in the user's cluster.

**Snippet 16.** Calculating cluster precision and recall for User 1

```
#initialize arrays for storing precision and recall for the three users
        precision = {}
        for i in range(0, 120): #0-40 – User 1's precision/recall, 41-80, User 2's precision/recall, etc…
           precision[i+1] = []
        recall = {}
        for i in range(0, 120):
           recall[i+1] = []

        image = 1
        count = 1

        #compare user1's clusters with the algorithm's clusters
        while count <= 40:#look at all images
           #find the cluster the image is in
           userkey = 0
           botkey = 0
           similar = 0
           for ukey in user1.keys():#find the user cluster that holds the given image
```

```
            if image in user1[ukey]:
                userkey = ukey
        for bkey in bot.keys():#find the system cluster that holds the given image
            if image in bot[bkey]:
                botkey = bkey
        #how many other images do the clusters containing the image have in common?
        #if they share very little, then the user placed their image in the "wrong"
        #cluster (or conversely, the system failed to meet user's expectations of how this
         #image should be categorized), and consequently the recall and precision will suffer
        similar = findsimilar(user1[userkey], bot[botkey])
        precision[count] = float(similar)/len(user1[userkey])
        recall[count] = float(similar)/len(bot[botkey])
        count += 1
        image += 1

    #calculate user1's average recall and precision
    recall1 = 0.0
    precision1 = 0.0
    for i in range(1, 41):
        recall1 += recall[i]
        precision1 += precision[i]
    recall1 = recall1/40.0 #divide by number of images to get the average recall and precision
    precision1 = precision1/40.0
```

The findsimilar function is used to check how many images the user-defined and program-generated clusters have in common. For example, suppose we are comparing the clusters containing Image 1. For user 1, the cluster that contains Image 1 looks like this:


For the system, the cluster containing Image 1 looks like this:


When the arrays containing these clusters are given to the findsimilar function, it finds that the two clusters have 6 images in common (Images 1, 3, 4, 8, 10, and 16), yielding a recall of 6/12 = 0.5 and a precision of 6/11 = 0.54545454545. These calculations are repeated for all forty images, and the forty recall and precision values are later divided by 40 to obtain the average recall and precision for the user.

**Snippet 17.** Function that finds how many images two clusters have in common.

```
#function that finds how many images two clusters (a user-defined cluster and a program-generated
#cluster) have in common
def findsimilar(user, bot):
    common = 0
    for i in range(0, 40):#iterate through all images
      #if an image is shared by both the user and system cluster, increment common
      if i+1 in user and i+1 in bot:
            common += 1
      return common
```

**Table 5.** Cluster precision and recall for each user.

Melanie Hsu
mlh2197

| Measure/User | 1 | 2 | 3 |
|---|---|---|---|
| Recall | 0.444716386555 | 0.466144957983 | 0.44618697479 |
| Precision | 0.488636363636 | 0.622916666667 | 0.51625 |

Overall, the recall for all three users was poor, indicating that the system really isn't making clusters that are relevant for users. This is unsurprising, considering that 2/3 users grouped the images based on content, while the system used color and texture. Again, this shows that while users are interested in color and textural information, they tend to make color and textural comparisons for similar classes of objects. After all, when people make color comparsions in real life, they usually compare the colors of objects, such as red iPhones to green iPhones, and not red iPhones to green cars.

What probably threw the algorithm off the most was the fact that all three users put items of different textural roughness, and to a lesser extent, images with weakly similar colors, into the same group. In the complete link clusters, all of the images are similar to each other in either or both color and texture. However, in some of User 1's clusters, the images are weakly related by color (for instance, yellow-orange and vivid orange items went into one cluster, and images with pale but colors went into another cluster), and sometimes not similar in texture whatsoever.

Users 2 and 3 made clusters based on the state of the objects in the image (ex. cut fruits, whole single foods), and some of these clusters were similar to the system's clusters because there happened to be a strong correlation between the state of the object and the object's color and/or texture. For instance, the system grouped the cut fruits together based on its observation that most of them are intermediately textured. This allowed the system's cluster of cut fruits to agree somewhat with User 2's and User 3's clsuters of cut fruit. However, the recall was thrown off because the system include many other simliarly-textured objects in this cluster, such as Images 28 and 33. Most users would probably not throw a carving of a cat in the same cluster as a bunch of food, further showing how unrealistic the system's clusters are.

Finally, the system does little to categorize images by the shapes of the objects in them. This contributed to its low recall agreement with User 3, who categorized objects based on shapes such as beadiness and stringiness, and User 2, who grouped images in a manner simliar to a grocery shopping list (whole foods in one section, multiple foods in another section, etc.). These are two very practical ways of clustering objects in real life; in order to appeal to users, the system would do well to take the shapes and states of objects into account. These two factors aren't easily gleaned from color and texture.

On average, the precision values are higher compared to the recall values, suggesting that the system is at least returning around half of the relevant clusters (from a user's standpoint). The precision value was highest for User 2, 0.622916666667, probably because the clusters they classified as "stringy" and "cut" just happened to share many images with the system's roughly textured, green object cluster and its intermediately-textured cluster. The "fractal," "liquid," and "miscellaneous" clusters also happened to match well against the system's clusters due to User 2's semantic definitions and the system's color and texture criteria arriving at similar clusters. Basically, the system and User 2 were making similar clusters for sometimes very different reasons, and if the data set consisted of different images, the precision might not have been so high.

As the low recall values show, having good precision isn't good enough either; the system should not only return more relevant than irrelevant results, but also return most of the relevant results. For instance, a user might want to find all pictures of round mochi, if the system only returns a subset of round mochi pictures, then it isn't giving a complete answer to the user. If the user is looking for pictures of plastic folders, then a system that returns both pictures of plastic folders and pictures of Macbook covers isn't answering the user's query correctly.

The precision for Users 1 and 3 is 0.488636363636 and 0.51625 respectively, which isn't that great considering that a precision of below 0.5 is worse than clustering images randomly. This again shows that

when it comes to clustering, this is a terrible system from a user's perspective: it doesn't consider criteria that many users value greatly, such as the object that's specifically depicted in the image. While the algorithm did a great job at the task it was given (rank images based on color and texture, cluster the images based on a mixture of color and texture), these tasks did not necessarily make the system useful for real people, who use a wider variety of criteria for classifying and ranking images. Some of these criteria, such as the content represented by the image, are incredibly hard to predict and implement algorithmically. Perhaps we would need to train the system to recognize different objects in images, just like how programs are trained on tagged datasets in natural language processing so that they can pick the correct parts of speech for words in a sentence.

**Part 5: Code**

To run part1 code: python color.py <name of directory containing the 40 images>
Below is the code for **color.py**:

```
import cv2
import numpy as np
import sys
import os
import imghdr #find type of image
from matplotlib import pyplot as plt
from PIL import Image
from PIL import ImageFont
from PIL import ImageDraw
from scipy import misc
import math
import random
from operator import attrgetter
import pylab
import matplotlib.cm as cm

#Melanie Hsu (mlh2197)
#This is the code for part1: comparing images based on color distributions. I used OpenCV to
#concatenate the table of septuples at the end, and to make the image of most similar and different
#images.

#round each pixel value so that it fits into one of the bins, and return the bin number it belongs to
def rnd(x):
        base=32#since using R, G, B results in a lot of bins (8^3 in this case), I used a larger base
        return int(base*math.floor(float(x)/base))/base

#calculate the L1_Norm between a pair of images
def L1norm(image1, image2):
        sum = 2*89*60
        distance = 0

        for i in range(0, 8):
           for j in range(0, 8):
                   for k in range(0, 8):
                           if i == 0 and j == 0 and k == 0:
                                   #remove black pixels from comparisons if the diff in black pixel count
                                   #between the two images is small; this is based on an assumption that
                                   #most images that meet this criteria both have backgrounds with similar
```

```
                                        #amounts of black pixels in them
                                        #ex. both have significant black backgrounds, very few black pixels, etc.
                                        #in contrast, if they have very different counts of black pixels then the
                                        #pictures should be rated as very different from each other
                                        if math.fabs(image1[0,0,0] - image2[0,0,0]) < 100:
                                                sum -= image1[0,0,0]#if we remove black pixels from
                                                #comparisons, must subtract the
                                                sum -= image2[0,0,0]#number of black pixels in these bins from
                                                #the total sum
                                                continue
                                distance += math.fabs(image1[i,j,k] - image2[i,j,k])
        norm = distance/sum
        return norm

if __name__ == '__main__':
        try:#pass in directory name, ex. Images
           folder = sys.argv[1]
           #print (folder)
        except:
           print ("Please pass directory name")

        files = os.listdir(folder)

        paths = []
        for stuff in files:#obtain the absolute path of each image
           paths.append("" + folder + "/" + stuff)#you have to change this based on where you're running this
        pics = []

        num = 0
        #test the image type and take just the ppms
        for image in paths:
           img = os.path.abspath(image)
           if imghdr.what(img) == 'ppm':
                        pics.append(img)
                        num += 1

        picture = {}
        counting = 0
        #make a RGB histogram for each image and save it to an array of histograms
        for pict in pics:
           hist = {}
           img = Image.open(pict)
           pix = img.load()
           for i in range(0,8):
                        for j in range(0,8):
                                for k in range(0,8):
                                        hist[(i, j, k)] = 0
           for i in range(0, 89):#iterate over dimensions of picture
                        for j in range(0, 60):
                                #print pix[i, j]
                                pixel = pix[i, j]
                                blue = rnd(pixel[0])#calculate the bin coordinates
                                green = rnd(pixel[1])
```

```
                    red = rnd(pixel[2])
                    hist[(blue, green, red)] += 1
    picture[counting] = hist
    counting += 1

#to calculate the similarity between each image and every other image, we must first
#initialize a dictionary of dictionaries to store the values
#in the end, dictionary will contain records of each image's similarity to every other image
#I'll be using this kind of technique again and again in later parts as well.
differences = {}
for i in range(len(picture)):#initialize a 2D array
    differences[i+1] = {}
colors = {}#a backup array, later on will be using this to calculate color ranks
for i in range(len(picture)):#initialize a 2D array
    colors[i+1] = {}
tally = 0
for i in range(len(picture)):
    for j in range(len(picture)):
            differences[i+1][j+1] = round(L1norm(picture[i], picture[j]), 5)# calc and store the L1 norms
            colors[i+1][j+1] = 1-differences[i+1][j+1]#save the similarity
            tally += 1

sept = {}
for i in range(0, 40):#initialize a 2D array
    sept[i+1] = {}#I started at the 1st index since the first image is Image 1, to avoid confusion

#compute the three most similar and most different images
for i in range(len(picture)):
    topfoe = i
    twofoe = i
    threefoe = i
    one = 0
    two = 0
    three = 0
    topfriend = i
    twofriend = i
    threefriend = i
    yi = 1
    er = 1
    san = 1
    for j in range(len(picture)):#find the top two and bottom two
            #logic to find the two most different images from each image
            if differences[i+1][j+1] > one:
                    two = one
                    twofoe = topfoe
                    one = differences[i+1][j+1]
                    topfoe = j+1
            elif one > differences[i+1][j+1] > two:
                    two = differences[i+1][j+1]
                    twofoe = j+1
            #logic to find the two most similar images to each image
            if 0 < differences[i+1][j+1] < yi:
                    twofriend = topfriend
```

```
                    er = yi
                    topfriend = j+1
                    yi = differences[i+1][j+1]
            if yi < differences[i+1][j+1] < er:
                    twofriend = j+1
                    er = differences[i+1][j+1]
    sept[i+1][0] = i+1
    sept[i+1][1] = topfriend
    sept[i+1][2] = twofriend
    sept[i+1][5] = twofoe
    sept[i+1][6] = topfoe

    for j in range(len(picture)):#find the third top and third bottom
            if differences[i+1][j+1] > three and j+1 != topfoe and j+1 != twofoe:
                    three = differences[i+1][j+1]
                    threefoe = j+1
            if 0 < differences[i+1][j+1] < san and j+1 != topfriend and j+1 != twofriend:
                    san = differences[i+1][j+1]
                    threefriend = j+1
    sept[i+1][3] = threefriend
    sept[i+1][4] = threefoe


#print a giant table consisting of septuples for each image
font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
big = Image.new('RGB', (89*7, 60*40))#make a big image to fit a septuple for each image
draw = ImageDraw.Draw(big)
row = 0
for i in range(len(sept)):
    count = 0
    for j in range(0, 7):
            if sept[i+1][j] < 10:#load in the image
                    path = os.path.abspath("Images/i0" + str(sept[i+1][j]) + ".jpg")
                    img = Image.open(path)
            else:
                    path = os.path.abspath("Images/i" + str(sept[i+1][j]) + ".jpg")
                    img = Image.open(path)
            big.paste(img, (89*count, 60*row))#place the thumbnail on the big image
            #write corresponding image number
            draw.text((89*count, 60*row), str(sept[i+1][j]), (255, 255, 255), font=font)
            draw.text((89*count, 30+60*row), str(1-differences[i+1][sept[i+1][j]]), (255, 255, 255),
                    font=font)#write similarity value
            count += 1
    row += 1
big.save("colors.bmp")#save the image table

compares = {}
for i in range(len(differences)):
    for j in range(len(differences)):
            for k in range(len(differences)):
                    for l in range(len(differences)):
                            #don't compare any image to itself
```

Melanie Hsu
mlh2197

```
                        if i != j and i != k and i != l and j != k and j != l and k != l:
                            #avg the difference between each image to every other image
                            average = (differences[i+1][j+1] + differences[i+1][k+1] +
                                differences[i+1][l+1] + differences[j+1][k+1]
                                + differences[j+1][l+1] + differences[k+1][l+1])/6
                            compares[(i+1,j+1,k+1,l+1)] = average
#find the four images that are most different from each other
tuple = (i,j,k,l)
biggest = 0
bigtuple = (0,0,0,0)
for tuple in compares:#compares already contains tuples of four images in each cell
    if compares[tuple] > biggest:
            biggest = compares[tuple]
            bigtuple = tuple
print biggest
print bigtuple
fourdif = Image.new('RGB', (89*4, 60))#logic to concatenate and print four most diff images
sketch = ImageDraw.Draw(fourdif)
tally = 0
for i in bigtuple:
    if i < 10:
            path = os.path.abspath("Images/i0" + str(i) + ".jpg")
            img = Image.open(path)
    else:
            path = os.path.abspath("Images/i" + str(i) + ".jpg")
            img = Image.open(path)
    fourdif.paste(img, (89*tally, 0))
    sketch.text((89*tally, 0), str(i), (255, 255, 255), font=font)
    tally += 1
fourdif.save("diffs.bmp")

#find the four images that are most similar to each other
tup = (i,j,k,l)
smallest = 1
smalltuple = (0,0,0,0)
for tuple in compares:
    if compares[tuple] < smallest:
            smallest = compares[tuple]
            smalltuple = tuple
print smallest
print smalltuple
foursam = Image.new('RGB', (89*4, 60))
scribble = ImageDraw.Draw(foursam)
head = 0
for i in smalltuple:
    if i < 10:
            path = os.path.abspath("Images/i0" + str(i) + ".jpg")
            img = Image.open(path)
    else:
            path = os.path.abspath("Images/i" + str(i) + ".jpg")
            img = Image.open(path)
    foursam.paste(img, (89*head, 0))
    scribble.text((89*head, 0), str(i), (255, 255, 255), font=font)
```

```
            head += 1
        foursam.save("similarities.bmp")

        #output the table to a file, which we will be using for parts 3 and 4
        f = open('colortable.txt', 'w')
        for i in range(len(differences)):
            array = differences[i+1]
            for i in range(len(array)):
                        diffs = 1-array[i+1]
                        f.write(str(diffs) + " ")
            f.write("\n")
            f.close()
```

To run part2 code: python texture.py <name of directory containing the 40 images>
Below is the code for **texture.py**:

```
import cv2
import numpy as np
import sys
import os
import imghdr #find type of image
from matplotlib import pyplot as plt
from PIL import Image
from PIL import ImageFont
from PIL import ImageDraw
from scipy import misc
import math
import random
from operator import attrgetter
import pylab
import matplotlib.cm as cm

#Melanie Hsu (mlh2197)
#This is the code for part 2, match images based on textures. I borrowed some of the code I wrote for
#part1. Again, I also used openCV to concatenate images.

#round each grayscale pixel so that it fits into one of the bins, and return the bin number it belongs to
def rnd(x):
        base=32
        return int(base*round(float(x)/base))/base

def sort(x):#range: -1320 to 1500, sort the laplacian pixels into bins
        base=100
        return int(base*round(float(x+1320)/base))/base#adjust so that smallest value = 0

#calculate the L1_Norm between every pair of images
def L1norm(image1, image2, black1, black2):
        sum = 2*89*60
        distance = 0

        #use the same zero subtraction used in color.py
        sum -= black1
        sum -= black2
```

```
        for i in range(0, 30):
            distance += math.fabs(image1[i] - image2[i])
        norm = distance/sum
        return norm

if __name__ == '__main__':
        try:#pass in directory name, ex. Images
            folder = sys.argv[1]
            #print (folder)
        except:
            print ("Please pass directory name")

        files = os.listdir(folder)

        paths = []
        for stuff in files:#obtain the absolute path of each image
            paths.append("" + folder + "/" + stuff)# change this depending on where you're running this
        pics = []

        num = 0
        #test the image type and take just the jpgs
        for image in paths:
            img = os.path.abspath(image)
            if imghdr.what(img) == 'ppm':
                        pics.append(img)
                        num += 1

        grayscale = {}
        counting = 0
        #make a grayscale image for each image, save it with a certain extension
        for pict in pics:
            gray = {}
            img = Image.open(pict)
            pix = img.load()
            grayimg = Image.new('L', (89, 60))
            for i in range(0, 89):#iterate over dimensions of picture
                        for j in range(0, 60):
                                    gray[i, j] = 0
                                    pixel = pix[i, j]
                                    gray[i,j] = (pixel[0] + pixel[1] + pixel[2])/3
                                    grayimg.putpixel((i, j), gray[i,j])#build the image pixel by pixel
            grayimg.save(str(counting+1) + "gray" + ".ppm")#save the grayscale image
            grayscale[counting] = grayimg
            counting += 1
        #get current working directory (since we saved grayscale images there)
        folder = os.path.dirname(os.path.realpath(__file__))
        files = os.listdir(folder)
        graypath = []
        for stuff in files:#get all the non-directory items
            if os.path.isdir(stuff):
                        continue
            else:
                        graypath.append(stuff)
```

Melanie Hsu
mlh2197

```
graypics = []
for image in graypath:#get all the grayscale images
   img = os.path.abspath(image)
   if imghdr.what(img) == 'pgm' and 'gray' in img:
            graypics.append(img)

count = 0
picarray = {}
black = {}#number of pixels that will be discarded as black in each image
#(need it to know how much to subtract when doing L1 calcs)
for i in range(1, 41):
   black[i] = 0#initialize the black pixel count
for picture in graypics:
   lap = {}
   hist = {}#histogram holder
   string = graypics[count].split("/")[-1]
   string = string.replace("gray.ppm", "")
   img = Image.open(picture)
   pix = img.load()
   for i in range(0, 30):#initialize histogram
                  hist[i] = 0
   for i in range(0, 89):
            for j in range(0, 60):
                  lap[i, j] = 0
                  pixel = pix[i, j]#grayscale pixel
                  rounded = rnd(pixel)#round to see which bin the grayscale pixel falls into
                  if rounded == 0:#don't calculate laplacian value for “black” pixels
                           black[int(string)] += 1
                           continue
                  multiplier = 0
                  subtractor = 0
                  if j > 0:
                           multiplier += 1#i, j-1 doable
                           subtractor += pix[i, j-1]
                  if j < 59:
                           multiplier += 1#i, j+1 doable
                           subtractor += pix[i, j+1]
                  if i > 0:
                           multiplier += 1#i-1, j doable
                           subtractor += pix[i-1, j]
                           if j > 0:
                                    multiplier += 1#i-1, j-1 doable
                                    subtractor += pix[i-1, j-1]
                           if j < 59:
                                    multiplier += 1#i-1, j+1 doable
                                    subtractor += pix[i-1, j+1]
                  if i < 88:
                           multiplier += 1#i+1, j doable
                           subtractor += pix[i+1, j]
                           if j > 0:
                                    multiplier += 1#i+1, j-1 doable
                                    subtractor += pix[i+1, j-1]
                           if j < 59:
```

```
                                    multiplier += 1#i+1, j+1 doable
                                    subtractor += pix[i+1, j+1]
                        lap[i, j] = multiplier*pixel - subtractor#each pixel value is 8*original pixel value
                        bin = sort(lap[i, j])#find which bin the Laplacian pixel belongs to
                        hist[bin] += 1 # then store it in the histogram
    picarray[int(string)] = hist #store the histogram for the image
    count += 1

#calculate the similarity between each image and every other image
differences = {}
for i in range(len(picarray)):#initialize a 2D array
    differences[i+1] = {}
tally = 0
for i in range(len(picarray)):
    for j in range(len(picarray)):#L1 norm calcs
            differences[i+1][j+1] = round(L1norm(picarray[i+1], picarray[j+1], black[i+1], black[j+1]), 5)
            tally += 1
sept = {}
for i in range(len(picarray)):#initialize a 2D array
    sept[i+1] = {}

#compute the three most similar and most different images
for i in range(len(picarray)):
    topfoe = i
    twofoe = i
    threefoe = i
    one = 0
    two = 0
    three = 0
    topfriend = i
    twofriend = i
    threefriend = i
    yi = 1
    er = 1
    san = 1
    for j in range(len(picarray)):#find the top two and bottom two
            #logic to find the two most different images from each image
            if differences[i+1][j+1] > one:
                    two = one
                    twofoe = topfoe
                    one = differences[i+1][j+1]
                    topfoe = j+1
            elif one > differences[i+1][j+1] > two:
                    two = differences[i+1][j+1]
                    twofoe = j+1
            #logic to find the two most similar images to each image
            if 0 < differences[i+1][j+1] < yi:
                    twofriend = topfriend
                    er = yi
                    topfriend = j+1
                    yi = differences[i+1][j+1]
            if yi < differences[i+1][j+1] < er:
                    twofriend = j+1
```

```
                    er = differences[i+1][j+1]
    sept[i+1][0] = i+1
    sept[i+1][1] = topfriend
    sept[i+1][2] = twofriend
    sept[i+1][5] = twofoe
    sept[i+1][6] = topfoe

    for j in range(len(picarray)):#find the third top and third bottom
            if differences[i+1][j+1] > three and j+1 != topfoe and j+1 != twofoe:
                    three = differences[i+1][j+1]
                    threefoe = j+1
            if 0 < differences[i+1][j+1] < san and j+1 != topfriend and j+1 != twofriend:
                    san = differences[i+1][j+1]
                    threefriend = j+1
    sept[i+1][3] = threefriend
    sept[i+1][4] = threefoe

#print a giant table consisting of septuples for each image
font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
big = Image.new('RGB', (89*7, 60*40))#make a big image to fit the septuples
draw = ImageDraw.Draw(big)
row = 0
for i in range(len(sept)):
    count = 0
    for j in range(0, 7):
            if sept[i+1][j] < 10:
                    path = os.path.abspath("Images/i0" + str(sept[i+1][j]) + ".jpg")
                    #print path
                    img = Image.open(path)
            else:
                    path = os.path.abspath("Images/i" + str(sept[i+1][j]) + ".jpg")
                    #print path
                    img = Image.open(path)
            big.paste(img, (89*count, 60*row))#place the thumbnail on the big image
            #write corresponding image number
            draw.text((89*count, 60*row), str(sept[i+1][j]), (255, 255, 255), font=font)
            draw.text((89*count, 30+60*row), str(1-differences[i+1][sept[i+1][j]]), (255, 255, 255),
                    font=font)#write similarity value
            count += 1
    row += 1
big.save("textures.bmp")#save the image table

compares = {}
for i in range(len(differences)):
    for j in range(len(differences)):
            for k in range(len(differences)):
                    for l in range(len(differences)):
                            #don't compare any image to itself
                            if i != j and i != k and i != l and j != k and j != l and k != l:
                                    #average the difference between each one to each other one
                                    #ab, ac, ad, bc, bd, cd
                                    average = (differences[i+1][j+1] + differences[i+1][k+1] +
                                            differences[i+1][l+1] + differences[j+1][k+1]
```

Melanie Hsu
mlh2197

```
                + differences[j+1][l+1] + differences[k+1][l+1])/6
            compares[(i+1,j+1,k+1,l+1)] = average

#find the four images that are most different from each other
tuple = (i,j,k,l)
biggest = 0
bigtuple = (0,0,0,0)
for tuple in compares:
    if compares[tuple] > biggest:
            biggest = compares[tuple]
            bigtuple = tuple
print biggest
print bigtuple
fourdif = Image.new('RGB', (89*4, 60))
sketch = ImageDraw.Draw(fourdif)
tally = 0
for i in bigtuple:
    if i < 10:
            path = os.path.abspath("Images/i0" + str(i) + ".jpg")
            img = Image.open(path)
    else:
            path = os.path.abspath("Images/i" + str(i) + ".jpg")
            img = Image.open(path)
    fourdif.paste(img, (89*tally, 0))
    sketch.text((89*tally, 0), str(i), (255, 255, 255), font=font)
    tally += 1
fourdif.save("diffstexture.bmp")

#find the four images that are most similar to each other
tup = (i,j,k,l)
smallest = 1
smalltuple = (0,0,0,0)
for tuple in compares:
    if compares[tuple] < smallest:
            smallest = compares[tuple]
            smalltuple = tuple
print smallest
print smalltuple
foursam = Image.new('RGB', (89*4, 60))
scribble = ImageDraw.Draw(foursam)
head = 0
for i in smalltuple:
    if i < 10:
            path = os.path.abspath("Images/i0" + str(i) + ".jpg")
            img = Image.open(path)
    else:
            path = os.path.abspath("Images/i" + str(i) + ".jpg")
            img = Image.open(path)
    foursam.paste(img, (89*head, 0))
    scribble.text((89*head, 0), str(i), (255, 255, 255), font=font)
    head += 1
foursam.save("similaritiestexture.bmp")
```

```
        #output the table to a file, for part3
        f = open('texturetable.txt', 'w')
        for i in range(len(differences)):
            array = differences[i+1]
            for i in range(len(array)):
                        diffs = 1-array[i+1]
                        f.write(str(diffs) + " ")
            f.write("\n")
            f.close()
```

To run part3 code: python colorxtexture.py
Below is the code for **colorxtexture.py**.

```
import cv2
import numpy as np
import sys
import os
import imghdr #find type of image
from matplotlib import pyplot as plt
from PIL import Image
from PIL import ImageFont
from PIL import ImageDraw
from scipy import misc
import math
import random
from operator import attrgetter
import pylab
import matplotlib.cm as cm

#Melanie Hsu (mlh2197)
#This is the code for part3: finding seven clusters using complete link, and then finding seven
#clusters using single link. I did not borrow anyone else's clustering code. Some code was taken
#from the color.py and texture.py that I wrote. I used OpenCV to concatenate the images for display.

#calculate the linear sum of an image
def linearsum(color, texture):
        r = 0.6
        linsum = r*float(texture) + (1.0-r)*float(color)
        return linsum

if __name__ == '__main__':
        colors = {}
        for i in range(0, 40):#initialize a 2D array to store the color similarities
            colors[i+1] = {}
        lines = {} #store the file contents

        with open("colortable.txt","r") as f:
            lines = f.readlines()

        for i in range(0, 40):#read the color similarity measures from the text file
            row = lines[i]
            row = row.split()
            for j in range(0, 40):
```

```
            colors[i+1][j+1] = row[j]

textures = {}
for i in range(0, 40):#initialize a 2D array to store the color similarities
   textures[i+1] = {}
line = {} #store the file contents

with open("texturetable.txt","r") as f:
   line = f.readlines()

for i in range(0, 40):#read the color similarity measures from the text file
   row = line[i]
   row = row.split()
   for j in range(0, 40):
            textures[i+1][j+1] = row[j]

linsum = {}
for i in range(0, 40):#initialize a 2D array to store the distance measures (1-linear sum)
   linsum[i+1] = {}
for i in range(0, 40):
   for j in range(0, 40):
            linsum[i+1][j+1] = 1-(linearsum(colors[i+1][j+1], textures[i+1][j+1]))

#This following section sorts the images into seven clusters based on complete link.

#pre-set the clusters
clusters = {}
for i in range(0, 40):#initialize a 2D array to store the distances (1-linear sum)
   clusters[i] = []
   clusters[i].append(i+1)

#reduce to 39 clusters
smallest = 1.0
first = 0
second = 0
for i in range(0, 40):
   for j in range(0, 40):
            if i == j:#don't compare an image to itself
                     continue
            one = linsum[i+1]#first image to compare
            two = linsum[j+1]#second image to compare
            difference = one[j+1]
            if difference < smallest:#logic to find the two closest images
                     first = i+1
                     second = j+1
                     smallest = difference
clusters[first-1].append(second)#add the contents of second cluster to the first cluster
clusters.pop(second-1)#remove the cluster that the second img belongs to
print clusters

#keep doing this until the clustering until number of clusters = 7
while len(clusters) > 7:
   mostfar = 1.0
```

```
    mostfar1 = []#will eventually store the
    mostfar2 = []#two clusters to be merged
    for keyA in clusters.keys():#compare each cluster to every other cluster
            for keyB in clusters.keys():
                    #key no longer in dict cause of merging with another cluster
                    if keyA not in clusters or keyB not in clusters:
                            continue
                    if keyA == keyB:#don't compare a cluster to itself
                            continue
                    gloop = clusters[keyA]
                    ploop = clusters[keyB]
                    farthest = 0.0
                    far1 = 0
                    far2 = 0
                    for k in range(len(gloop)):#compare the farthest images in each cluster
                            for l in range(len(ploop)):
                                    one = linsum[gloop[k]]
                                    two = linsum[ploop[l]]
                                    difference = one[ploop[l]]
                                    if difference > farthest:
                                            far1 = gloop[k]
                                            far2 = ploop[l]
                                            #this ends up as the distance measure between these
                                            #two clusters
                                            farthest = difference
                                    if farthest < mostfar:#find the two nearest clusters (to merge)
                                            mostfar = farthest
                                            #these end up as the two clusters to merge
                                            mostfar1 = keyA
                                            mostfar2 = keyB
    A = clusters[mostfar1]
    B = clusters[mostfar2]
    for i in range(len(B)):
            A.append(B[i])#append contents of cluster B to cluster A
    clusters.pop(mostfar2)#remove cluster B from the dictionary

#find the longest cluster (the cluster with the most number of images in it)
longest = 0
for theKey in clusters.keys():
    if len(clusters[theKey]) > longest:
            longest = len(clusters[theKey])

#make a giant image of all the clusters together
font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
complete = Image.new('RGB', (89*longest, 60*7))#image width defined by the largest cluster
draw = ImageDraw.Draw(complete)
row = 0
for theKey in clusters.keys():#for each cluster
    for i in range(len(clusters[theKey])):
            if clusters[theKey][i] < 10:#get the image
                    path = os.path.abspath("Images/i0" + str(clusters[theKey][i]) + ".jpg")
                    img = Image.open(path)
            else:
```

```
                path = os.path.abspath("Images/i" + str(clusters[theKey][i]) + ".jpg")
                img = Image.open(path)
            complete.paste(img, (89*i, 60*row))#place the thumbnail on the big image
            draw.text((89*i, 60*row), str(clusters[theKey][i]), (255, 255, 255), font=font)
    row += 1
complete.save("complete.bmp")#save the cluster image for complete link

#write the complete link cluster information to a file, needed for part4 comparisons
f = open('colortexture.txt', 'w')
for theKey in clusters.keys():
    array = clusters[theKey]
    for i in range(len(array)):
            f.write(str(array[i]) + " ")
    f.write("\n")
f.close()

#The following section sorts the images into seven clusters based on single link

#pre-set the clusters, same as in the complete link code
bunches = {}
for i in range(0, 40):#initialize a 2D array to store the distances (1-linear sum)
    bunches[i] = []
    bunches[i].append(i+1)

#reduce to 39 clusters; essentially done the same way as the complete link method
smallest = 1.0
first = 0
second = 0
for i in range(0, 40):
    for j in range(0, 40):
            if i == j:#don't compare an image to itself
                continue
            one = linsum[i+1]#first image to compare
            two = linsum[j+1]#second image to compare
            difference = one[j+1]
            if difference < smallest:
                first = i+1
                second = j+1
                smallest = difference
bunches[first-1].append(second)
bunches.pop(second-1)

#keep doing this until the clustering until number of clusters = 7
while len(bunches) > 7:
    mostfar = 1.0
    mostfar1 = []
    mostfar2 = []
    for keyA in bunches.keys():#compare each cluster to every other cluster
            for keyB in bunches.keys():
                #key gone cause of merging with another cluster
                if keyA not in bunches or keyB not in bunches:
                    continue
                if keyA == keyB:#don't compare a cluster to itself
```

```
                        continue
                    gloop = bunches[keyA]
                    ploop = bunches[keyB]
                    nearest = 1.0
                    far1 = 0
                    far2 = 0
                    for k in range(len(gloop)):#compare the farthest images in each cluster
                        for l in range(len(ploop)):
                            one = linsum[gloop[k]]
                            two = linsum[ploop[l]]
                            difference = one[ploop[l]]
                            if difference < nearest:
                                far1 = gloop[k]
                                far2 = ploop[l]
                                #this ends up as the distance measure between these
                                #two clusters
                                nearest = difference
                            if nearest < mostfar:#find the two nearest clusters (to merge)
                                mostfar = nearest
                                #these end up as the two clusters to merge
                                mostfar1 = keyA
                                mostfar2 = keyB
        A = bunches[mostfar1]
        B = bunches[mostfar2]
        for i in range(len(B)):
            A.append(B[i])
        bunches.pop(mostfar2)

    #find the longest cluster
    longest = 0
    for theKey in bunches.keys():
        if len(bunches[theKey]) > longest:
            longest = len(bunches[theKey])

    #make a giant image of all the clusters together
    font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
    single = Image.new('RGB', (89*longest, 60*7))#image width defined by the largest cluster
    draw = ImageDraw.Draw(single)
    row = 0
    for theKey in bunches.keys():#for each cluster
        for i in range(len(bunches[theKey])):
            if bunches[theKey][i] < 10:#get the image
                path = os.path.abspath("Images/i0" + str(bunches[theKey][i]) + ".jpg")
                img = Image.open(path)
            else:
                path = os.path.abspath("Images/i" + str(bunches[theKey][i]) + ".jpg")
                img = Image.open(path)
            single.paste(img, (89*i, 60*row))#place the thumbnail on the big image
            draw.text((89*i, 60*row), str(bunches[theKey][i]), (255, 255, 255), font=font)
        row += 1
        single.save("single.bmp")#save the cluster image for single link
```

To run part4 code: python user.py

Melanie Hsu
mlh2197

Below is the code for **user.py**.

```
#idea: you can do the part 3 ranking similar to how precision and recall are done in NLP
#for parts 1 and 2, compare ranks, ex. was their pick my 2nd, 3rd, 4th choice? then subtract
#accuracy points for each position you fall behind...

import cv2
import numpy as np
import sys
import os
import imghdr #find type of image
from matplotlib import pyplot as plt
from PIL import Image
from PIL import ImageFont
from PIL import ImageDraw
from scipy import misc
import math
import random
from operator import attrgetter
import pylab
import matplotlib.cm as cm

#function that finds how many images two clusters (a user-defined cluster and a program-generated
#cluster) have in common
def findsimilar(user, bot):
	common = 0
	for i in range(0, 40):#iterate through all images, check if they're in user and bot clusters
	  if i+1 in user and i+1 in bot:
			common += 1
	return common

if __name__ == '__main__':
	colors = {}
	for i in range(0, 40):#initialize a 2D array to store the color similarities
	  colors[i+1] = {}
	lines = {} #store the file contents

	with open("colortable.txt","r") as f:
	  lines = f.readlines()

	for i in range(0, 40):#read the color similarity measures from the text file
	  row = lines[i]
	  row = row.split()
	  for j in range(0, 40):
			colors[i+1][j+1] = row[j]

	textures = {}
	for i in range(0, 40):#initialize a 2D array to store the color similarities
	  textures[i+1] = {}
	line = {} #store the file contents

	with open("texturetable.txt","r") as f:
	  line = f.readlines()
```

Melanie Hsu
mlh2197

```
for i in range(0, 40):#read the color similarity measures from the text file
    row = line[i]
    row = row.split()
    for j in range(0, 40):
            textures[i+1][j+1] = row[j]

colorranks = {}#initialize array to place the image rankings made by color.py
for i in range(0, 40):
    colorranks[i+1] = []

#for each of the images, reorder the images from most to least similar in color
#I made the assumption that the user also ranked images from most to least similar,
#and that we are comparing our most simliar image to their most similar image
i = 1
while len(colors) > 0:
    while len(colors[i]) > 1:#go through each image's array
            biggest = 0.0#find the most similar image out of the images left in colors
            bigkey = 0
            color = colors[i]
            #print color
            for A in color.keys():
                    if A == i:#don't compare image with itself
                            continue
                    if color[A] > biggest:
                            biggest = color[A]
                            bigkey = A
            #found the most similar image
            colorranks[i].append(bigkey)#we only care about the number of the image
            colors[i].pop(bigkey)#remove the smallest from the colors array
    if len(colors[i]) == 1:
            colors.pop(i)#done with the image
            i += 1

textureranks = {}#initialize array to place the texture rankings made by texture.py
for i in range(0, 40):
    textureranks[i+1] = []

i = 1
while len(textures) > 0:
    while len(textures[i]) > 1:#go through each image's array
            biggest = 0.0#find the most similar image out of the images left in texture
            bigkey = 0
            text = textures[i]
            for A in text.keys():
                    if A == i:#don't compare image with itself
                            continue
                    if text[A] > biggest:
                            biggest = text[A]
                            bigkey = A
            #found the most similar image
            textureranks[i].append(bigkey)#we only care about the number of the image
            textures[i].pop(bigkey)#remove the smallest from the textures array
```

```
    if len(textures[i]) == 1:
            textures.pop(i)#done with the image
            i += 1

colorsim1 = [0 for i in range(40)]#user 1 most similar color ratings
colorsim2 = [0 for i in range(40)]#user 2 most similar color ratings
colorsim3 = [0 for i in range(40)]#user 3 most similar color ratings
colordiff1 = [0 for i in range(40)]#user 1 most different color ratings
colordiff2 = [0 for i in range(40)]#user 2 most different color ratings
colordiff3 = [0 for i in range(40)]#user 3 most different color ratings
line = {} #store the file contents

#read in files containing the three user similarity and difference measures for color
with open("user1colorsim.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    colorsim1[j] = row[j]

with open("user2colorsim.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    colorsim2[j] = row[j]

with open("user3colorsim.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    colorsim3[j] = row[j]

with open("user1colordiff.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    colordiff1[j] = row[j]

with open("user2colordiff.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    colordiff2[j] = row[j]
```

Melanie Hsu
mlh2197

```
with open("user3colordiff.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    colordiff3[j] = row[j]

#ranking distances for color for each user
csimdist1 = 0
csimdist2 = 0
csimdist3 = 0
cdiffdist1 = 0
cdiffdist2 = 0
cdiffdist3 = 0
#start with a distance of 0, and add 1 for the distance of each rank
#at the end, divide by 40 to get the average rank distance for each user
total = float(40)
i = 0
while i < 40:#for each image
    #for similarities, compare from front of array to back
    colors = colorranks[i+1]
    distance = 0#how many ranks apart is the user's most similar and my most similar image?
    for j in range(len(colors)):
            if int(colors[j]) != int(colorsim1[i]):
                    distance += 1
                    continue
            #the larger the distance between ranks, the more it will affect the total distance measure
            else:
                    csimdist1 += distance
                    break#match!
    i += 1
user1colorsims = (csimdist1)/total

i = 0
while i < 40:#for each image
    #for similarities, compare from front of array to back
    colors = colorranks[i+1]
    distance = 0
    for j in range(len(colors)):
            if int(colors[j]) != int(colorsim2[i]):
                    distance += 1
                    continue
            else:
                    csimdist2 += distance
                    break#match!
    i += 1
user2colorsims = (csimdist2)/total
print user2colorsims

i = 0
while i < 40:#for each image
    #for similarities, compare from front of array to back
```

```
    colors = colorranks[i+1]
    distance = 0
    for j in range(len(colors)):
            if int(colors[j]) != int(colorsim3[i]):
                    distance += 1
                    continue
            else:
                    csimdist3 += distance
                    break#match!
    i += 1
user3colorsims = (csimdist3)/total
print user3colorsims

i = 0#for differences, do the calc from the back of the array to the front
while i < 40:#for each image
    #for similarities, compare from front of array to back
    colors = colorranks[i+1]
    distance = 0#how many ranks apart is the user's most similar and my most similar image?
    j = 38
    while j >= 0:
            if int(colors[j]) != int(colordiff1[i]):
                    distance += 1
                    j -= 1
                    continue
            #the larger the distance between ranks, the more it will affect the total distance measure
            else:
                    cdiffdist1 += distance
                    break#match!
    i += 1
user1colordiffs = (cdiffdist1)/total

i = 0#for differences, do the calc from the back of the array to the front
while i < 40:#for each image
    #for similarities, compare from front of array to back
    colors = colorranks[i+1]
    distance = 0#how many ranks apart is the user's most similar and my most similar image?
    j = 38
    while j >= 0:
            if int(colors[j]) != int(colordiff2[i]):
                    distance += 1
                    j -= 1
                    continue
            else:
                    cdiffdist2 += distance
                    break#match!
    i += 1
user2colordiffs = (cdiffdist2)/total

i = 0#for differences, do the calc from the back of the array to the front
while i < 40:#for each image
    #for similarities, compare from front of array to back
    colors = colorranks[i+1]
    distance = 0#how many ranks apart is the user's most similar and my most similar image?
```

```
    j = 38
    while j >= 0:
            if int(colors[j]) != int(colordiff3[i]):
                    distance += 1
                    j -= 1
                    continue
            else:
                    cdiffdist3 += distance
                    break#match!
    i += 1
user3colordiffs = (cdiffdist3)/total

texturesim1 = [0 for i in range(40)]#user 1 most similar color ratings
texturesim2 = [0 for i in range(40)]#user 2 most similar color ratings
texturesim3 = [0 for i in range(40)]#user 3 most similar color ratings
texturediff1 = [0 for i in range(40)]#user 1 most different color ratings
texturediff2 = [0 for i in range(40)]#user 2 most different color ratings
texturediff3 = [0 for i in range(40)]#user 3 most different color ratings
line = {} #store the file contents

#read in files containing the three user similarity and difference measures for texture
with open("user1texturesim.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    texturesim1[j] = row[j]

with open("user2texturesim.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    texturesim2[j] = row[j]

with open("user3texturesim.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    texturesim3[j] = row[j]

with open("user1texturediff.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    texturediff1[j] = row[j]
```

Melanie Hsu
mlh2197

```
with open("user2texturediff.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    texturediff2[j] = row[j]

with open("user3texturediff.txt","r") as f:
    line = f.readlines()

row = line[0]
row = row.split()
for j in range(0, 40):
    texturediff3[j] = row[j]

#ranking distances for color for each user
tsimdist1 = 0
tsimdist2 = 0
tsimdist3 = 0
tdiffdist1 = 0
tdiffdist2 = 0
tdiffdist3 = 0

#calculate the difference between ranks using a variant of the Spearman correlation
#I adjusted the formula so that if the user and I agreed on all rankings, the correlation would be 1.0,
#and if we disagreed greatly on all rankings, the correlation would be 0.
total = float(40)
i = 0
while i < 40:#for each image
    #for similarities, compare from front of array to back
    texture = textureranks[i+1]
    distance = 0#how many ranks apart is the user's most similar and my most similar image?
    for j in range(len(texture)):
            if int(texture[j]) != int(texturesim1[i]):
                    distance += 1
                    continue
            else:
                    tsimdist1 += distance
                    break#match!
    i += 1
user1texturesims = (tsimdist1)/total

i = 0
while i < 40:#for each image
    #for similarities, compare from front of array to back
    texture = textureranks[i+1]
    distance = 0
    for j in range(len(texture)):
            if int(texture[j]) != int(texturesim2[i]):
                    distance += 1
                    continue
            else:
```

```
                        tsimdist2 += distance
                        break#match!
    i += 1
user2texturesims = (tsimdist2)/total

i = 0
while i < 40:#for each image
    #for similarities, compare from front of array to back
    texture = textureranks[i+1]
    distance = 0
    for j in range(len(texture)):
            if int(texture[j]) != int(texturesim3[i]):
                        distance += 1
                        continue
            else:
                        tsimdist3 += distance
                        break#match!
    i += 1
user3texturesims = (tsimdist3)/total

i = 0#for differences, do the calc from the back of the array to the front
while i < 40:#for each image
    #for similarities, compare from front of array to back
    texture = textureranks[i+1]
    distance = 0#how many ranks apart is the user's most similar and my most similar image?
    j = 38
    while j >= 0:
            if int(texture[j]) != int(texturediff1[i]):
                        distance += 1
                        j -= 1
                        continue
            else:
                        tdiffdist1 += distance
                        break#match!
    i += 1
user1texturediffs = (tdiffdist1)/total

i = 0#for differences, do the calc from the back of the array to the front
while i < 40:#for each image
    #for similarities, compare from front of array to back
    texture = textureranks[i+1]
    distance = 0#how many ranks apart is the user's most similar and my most similar image?
    j = 38
    while j >= 0:
            if int(texture[j]) != int(texturediff2[i]):
                        distance += 1
                        j -= 1
                        continue
            else:
                        tdiffdist2 += distance
                        break#match!
    i += 1
user2texturediffs = (tdiffdist2)/total
```

Melanie Hsu
mlh2197

```
i = 0#for differences, do the calc from the back of the array to the front
while i < 40:#for each image
   #for similarities, compare from front of array to back
   texture = textureranks[i+1]
   distance = 0#how many ranks apart is the user's most similar and my most similar image?
   j = 38
   while j >= 0:
            if int(texture[j]) != int(texturediff3[i]):
                     distance += 1
                     j -= 1
                     continue
            else:
                     tdiffdist3 += distance
                     break#match!
   i += 1
user3texturediffs = (tdiffdist3)/total

sept = {}
for i in range(0, 40):#initialize a 2D array for storing user outputs for color
   sept[i+1] = {}

for i in range(0, 40):
   sept[i+1][0] = i+1
   sept[i+1][1] = int(texturesim1[i])
   sept[i+1][2] = int(texturesim2[i])
   sept[i+1][3] = int(texturesim3[i])
   sept[i+1][4] = int(texturediff1[i])
   sept[i+1][5] = int(texturediff2[i])
   sept[i+1][6] = int(texturediff3[i])

#print an image for each user's outputs for texture
font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
big = Image.new('RGB', (89*7, 60*40))#make a big image to fit a septuple for each image
draw = ImageDraw.Draw(big)
row = 0
for i in range(len(sept)):
   count = 0
   for j in range(0, 7):
            if sept[i+1][j] < 10:#load in the image
                     path = os.path.abspath("Images/i0" + str(sept[i+1][j]) + ".jpg")
                     img = Image.open(path)
            else:
                     path = os.path.abspath("Images/i" + str(sept[i+1][j]) + ".jpg")
                     img = Image.open(path)
            big.paste(img, (89*count, 60*row))#place the thumbnail on the big image
            draw.text((89*count, 60*row), str(sept[i+1][j]), (255, 255, 255), font=font)
            count += 1
   row += 1
big.save("usertexture.bmp")#save the image table

octo = {}
for i in range(0, 40):#initialize a 2D array for storing user outputs for color
```

```
    octo[i+1] = {}

for i in range(0, 40):
    octo[i+1][0] = i+1
    octo[i+1][1] = int(colorsim1[i])
    octo[i+1][2] = int(colorsim2[i])
    octo[i+1][3] = int(colorsim3[i])
    octo[i+1][4] = int(colordiff1[i])
    octo[i+1][5] = int(colordiff2[i])
    octo[i+1][6] = int(colordiff3[i])

#print an image for each user's outputs for color
font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
big = Image.new('RGB', (89*7, 60*40))#make a big image to fit a septuple for each image
draw = ImageDraw.Draw(big)
row = 0
for i in range(len(sept)):
    count = 0
    for j in range(0, 7):
            if octo[i+1][j] < 10:#load in the image
                    path = os.path.abspath("Images/i0" + str(octo[i+1][j]) + ".jpg")
                    img = Image.open(path)
            else:
                    path = os.path.abspath("Images/i" + str(octo[i+1][j]) + ".jpg")
                    img = Image.open(path)
            big.paste(img, (89*count, 60*row))#place the thumbnail on the big image
            draw.text((89*count, 60*row), str(octo[i+1][j]), (255, 255, 255), font=font)
            count += 1
    row += 1
big.save("usercolor.bmp")#save the image table

#for part 3, store array of seven clusters
user1 = {}#initialize array to place the seven clusters for user1
for i in range(0, 7):
    user1[i+1] = []

user2 = {}#initialize array to place the seven clusters for user2
for i in range(0, 7):
    user2[i+1] = []

user3 = {}#initialize array to place the seven clusters for user3
for i in range(0, 7):
    user3[i+1] = []

bot = {}#initialize array to place the seven clusters from part3
for i in range(0, 7):
    bot[i+1] = []
#save two copies of the bot cluster array, cause we're going to be popping things off them
bot1 = {}
for i in range(0, 7):
    bot1[i+1] = []
bot2 = {}
for i in range(0, 7):
```

```
    bot2[i+1] = []

#read in the seven clusters for each user
with open("cluster1.txt","r") as f:
    line = f.readlines()
for i in range(len(line)):#read the clusters from the text file
    row = line[i]
    row = row.split()
    for j in range(len(row)):
            user1[i+1].append(int(row[j]))

with open("cluster2.txt","r") as f:
    line = f.readlines()
for i in range(len(line)):#read the clusters from the text file
    row = line[i]
    row = row.split()
    for j in range(len(row)):
            user2[i+1].append(int(row[j]))

with open("cluster3.txt","r") as f:
    line = f.readlines()
for i in range(len(line)):#read the clusters from the text file
    row = line[i]
    row = row.split()
    for j in range(len(row)):
            user3[i+1].append(int(row[j]))

with open("colortexture.txt","r") as f:
    line = f.readlines()
for i in range(len(line)):#read the clusters from the text file
    row = line[i]
    row = row.split()
    for j in range(len(row)):
            bot[i+1].append(int(row[j]))
            bot1[i+1].append(int(row[j]))
            bot2[i+1].append(int(row[j]))

#find the longest cluster for user1
longest = 0
for theKey in user1.keys():
    if len(user1[theKey]) > longest:
            longest = len(user1[theKey])

#make a giant image of all the clusters for user1
font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
single = Image.new('RGB', (89*longest, 60*7))#image width defined by the largest cluster
draw = ImageDraw.Draw(single)
row = 0
for theKey in user1.keys():#for each cluster
    for i in range(len(user1[theKey])):
            if user1[theKey][i] < 10:#get the image
                    path = os.path.abspath("Images/i0" + str(user1[theKey][i]) + ".jpg")
                    img = Image.open(path)
```

```
            else:
                path = os.path.abspath("Images/i" + str(user1[theKey][i]) + ".jpg")
                img = Image.open(path)
            single.paste(img, (89*i, 60*row))#place the thumbnail on the big image
            draw.text((89*i, 60*row), str(user1[theKey][i]), (255, 255, 255), font=font)
    row += 1
single.save("cluster1.bmp")#save the cluster image for user1

#find the longest cluster for user2
longest = 0
for theKey in user2.keys():
    if len(user2[theKey]) > longest:
            longest = len(user2[theKey])

#make a giant image of all the clusters for user2
font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
single = Image.new('RGB', (89*longest, 60*7))#image width defined by the largest cluster
draw = ImageDraw.Draw(single)
row = 0
for theKey in user2.keys():#for each cluster
    for i in range(len(user2[theKey])):
            if user2[theKey][i] < 10:#get the image
                path = os.path.abspath("Images/i0" + str(user2[theKey][i]) + ".jpg")
                img = Image.open(path)
            else:
                path = os.path.abspath("Images/i" + str(user2[theKey][i]) + ".jpg")
                img = Image.open(path)
            single.paste(img, (89*i, 60*row))#place the thumbnail on the big image
            draw.text((89*i, 60*row), str(user2[theKey][i]), (255, 255, 255), font=font)
    row += 1
single.save("cluster2.bmp")#save the cluster image for user2

#find the longest cluster for user3
longest = 0
for theKey in user3.keys():
    if len(user3[theKey]) > longest:
            longest = len(user3[theKey])

#make a giant image of all the clusters for user3
font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
single = Image.new('RGB', (89*longest, 60*7))#image width defined by the largest cluster
draw = ImageDraw.Draw(single)
row = 0
for theKey in user3.keys():#for each cluster
    for i in range(len(user3[theKey])):
            if user3[theKey][i] < 10:#get the image
                path = os.path.abspath("Images/i0" + str(user3[theKey][i]) + ".jpg")
                img = Image.open(path)
            else:
                path = os.path.abspath("Images/i" + str(user3[theKey][i]) + ".jpg")
                img = Image.open(path)
            single.paste(img, (89*i, 60*row))#place the thumbnail on the big image
            draw.text((89*i, 60*row), str(user3[theKey][i]), (255, 255, 255), font=font)
```

```
    row += 1
single.save("cluster3.bmp")#save the cluster image for user3

#assume that the clusters generated by the program is the gold standard, what is the
#recall and precision of the three users' clusters?
#recall = number of images shared by the bot and user cluster/number of images in the bot cluster
#precision = number of images shared by the bot and user cluster/number of images in the user
#cluster

#initialize arrays for storing precision and recall for the users
precision = {}
for i in range(0, 120):
    precision[i+1] = []
recall = {}
for i in range(0, 120):
    recall[i+1] = []

image = 1
count = 1

#compare user1 with the algorithm
while count <= 40:#look at all images
    #find the cluster the image is in
    userkey = 0
    botkey = 0
    similar = 0
    for ukey in user1.keys():#find image in the cluster
            if image in user1[ukey]:
                    userkey = ukey
    for bkey in bot.keys():
            if image in bot[bkey]:
                    botkey = bkey
    #how many other images do the clusters containing the image have in common?
    #if they share very little, then the user placed their image in the "wrong"
    #cluster, and the recall and precision will suffer
    similar = findsimilar(user1[userkey], bot[botkey])
    precision[count] = float(similar)/len(user1[userkey])
    recall[count] = float(similar)/len(bot[botkey])
    count += 1
    image += 1

#calculate user1's average recall and precision
recall1 = 0.0
precision1 = 0.0
for i in range(1, 41):
    recall1 += recall[i]
    precision1 += precision[i]
recall1 = recall1/40.0
precision1 = precision1/40.0
print "USER1 recall: " + str(recall1)
print "USER1 precision: " + str(precision1)

image = 1
```

Melanie Hsu
mlh2197

```
#compare user2 with the algorithm
while count <= 80:#look at all images
   #find the cluster the image is in
   userkey = 0
   botkey = 0
   similar = 0
   for ukey in user2.keys():#find image in the cluster
            if image in user2[ukey]:
                     userkey = ukey
   for bkey in bot1.keys():
            if image in bot1[bkey]:
                     botkey = bkey
   #how many other images do the clusters containing the image have in common?
   #if they share very little, then the user placed their image in the "wrong"
   #cluster, and the recall and precision will suffer
   similar = findsimilar(user2[userkey], bot1[botkey])
   precision[count] = float(similar)/len(user2[userkey])
   recall[count] = float(similar)/len(bot1[botkey])
   count += 1
   image += 1

#calculate user1's average recall and precision
recall2 = 0.0
precision2 = 0.0
for i in range(41, 81):
   recall2 += recall[i]
   precision2 += precision[i]
recall2 = recall2/40.0
precision2 = precision2/40.0
print "USER2 recall: " + str(recall2)
print "USER2 precision: " + str(precision2)

image = 1
#compare user2 with the algorithm
while count <= 120:#look at all images
   #find the cluster the image is in
   userkey = 0
   botkey = 0
   similar = 0
   for ukey in user3.keys():#find image in the cluster
            if image in user3[ukey]:
                     userkey = ukey
   for bkey in bot2.keys():
            if image in bot2[bkey]:
                     botkey = bkey
   #how many other images do the clusters containing the image have in common?
   #if they share very little, then the user placed their image in the "wrong"
   #cluster, and the recall and precision will suffer
   similar = findsimilar(user3[userkey], bot2[botkey])
   precision[count] = float(similar)/len(user3[userkey])
   recall[count] = float(similar)/len(bot2[botkey])
   count += 1
   image += 1
```

Melanie Hsu
mlh2197

```
#calculate user1's average recall and precision
recall3 = 0.0
precision3 = 0.0
for i in range(81, 121):
   recall3 += recall[i]
   precision3 += precision[i]
recall3 = recall3/40.0
precision3 = precision3/40.0
print "USER3 recall: " + str(recall3)
print "USER3 precision: " + str(precision3)

#make a giant image of all the images and their ranked arrays
font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
single = Image.new('RGB', (89*40, 60*40))
draw = ImageDraw.Draw(single)
row = 0
first = True
for theKey in colorranks.keys():#for each cluster
   if theKey < 10:#get the image
            path = os.path.abspath("Images/i0" + str(theKey) + ".jpg")
            img = Image.open(path)
   else:
            path = os.path.abspath("Images/i" + str(theKey) + ".jpg")
            img = Image.open(path)
   single.paste(img, (89*0, 60*row))#place the thumbnail on the big image
   draw.text((89*0, 60*row), str(theKey), (255, 255, 255), font=font)
   for i in range(len(colorranks[theKey])):
            if colorranks[theKey][i] < 10:#get the image
                     path = os.path.abspath("Images/i0" + str(colorranks[theKey][i]) + ".jpg")
                     img = Image.open(path)
            else:
                     path = os.path.abspath("Images/i" + str(colorranks[theKey][i]) + ".jpg")
                     img = Image.open(path)
            single.paste(img, (89*i+89, 60*row))#place the thumbnail on the big image
            draw.text((89*i+89, 60*row), str(colorranks[theKey][i]), (255, 255, 255), font=font)
   row += 1
   first = True #print the given image
single.save("color-rank.bmp")#save the cluster image for the color ranks

#make a giant image of all the images and their ranked arrays
font = ImageFont.truetype("MouseMemoirs-Regular.ttf", 25)#load in your own font
single = Image.new('RGB', (89*40, 60*40))
draw = ImageDraw.Draw(single)
row = 0
first = True
for theKey in textureranks.keys():#for each cluster
   if theKey < 10:#get the image
            path = os.path.abspath("Images/i0" + str(theKey) + ".jpg")
            img = Image.open(path)
   else:
            path = os.path.abspath("Images/i" + str(theKey) + ".jpg")
            img = Image.open(path)
```

```
    single.paste(img, (89*0, 60*row))#place the thumbnail on the big image
    draw.text((89*0, 60*row), str(theKey), (255, 255, 255), font=font)
    for i in range(len(textureranks[theKey])):
            if textureranks[theKey][i] < 10:#get the image
                    path = os.path.abspath("Images/i0" + str(textureranks[theKey][i]) + ".jpg")
                    img = Image.open(path)
            else:
                    path = os.path.abspath("Images/i" + str(textureranks[theKey][i]) + ".jpg")
                    img = Image.open(path)
            single.paste(img, (89*i+89, 60*row))#place the thumbnail on the big image
            draw.text((89*i+89, 60*row), str(textureranks[theKey][i]), (255, 255, 255), font=font)
    row += 1
    first = True #print the given image
single.save("texture-rank.bmp")#save the cluster image for the color ranks
```