Columbia University
COMS W4735y
Visual Interfaces to Computers
Assignment 2

Danilo Faria de Lima
df2553@columbia.edu

Spring 2015

Contents

1 Introduction

I will present this document in the following manner. I start presenting the survey, in section 2, explaining how I conducted it and what the results were. Next, in section 3 I talk about how I implemented gross color matching, what implementation decisions I've made and parameters I've used, what the output was (the 3 most alike images and unlike images to each of the 40 input images), what I did to find the 4 most alike and unalike images and how my implementation compares to the survey results. I do the same thing in section 4, but for gross texture matching. In section 5 I show how I combined gross color and texture matching, how I implemented single and complete link clustering and how my image groups compare to the ones of the survey. To help the grader, all images are shown with the identifying number on them. My project was implemented in MATLAB. Notice that I did the creative step at the same time I did all the other steps.

2 The survey

For the survey portion of the assignment, I interviewed 3 patient friends. Let's call them Rachel (R), Joseph (J) and Oliver (O). The first question was as follows: "For each of the 40 images provided, pick the most similar image in color and the least similar image in color". The results are shown in Figure 1, split in two halves. In the figure the column labeled "img" displays sequentially the images the participants had to compare, "R1" is for Rachel's choice of most similar image, "R0" is for Rachel's choice of least similar image, and "J1", "J0", "O1"and "O0" are similarly defined for Joseph and Oliver respectively.

The second question was the same as in the first, but the participants were asked to compare textures, not colors. The results are shown in Figure 2 and the labels have the same meaning as in Figure 1.

The third task was to group the 40 images into 7 groups by similarity considering both color and texture. Rachel's, Joseph's and Oliver's answers can be viewed in Figures 3 to 5. In each of those images, each row of images is a group (cluster).

The survey was conducted online. For tasks 1 and 2, the participants were given a table with the 40 images (bit.ly/1GbX32N) and a link to a Google doc with a form to fill with their answers.

For task 3, the participants were given the link to a web interface (bit.ly/1FqXR5N) in which they could drag and drop each of the 40 images into of 7 boxes, then they would have to submit a screenshot of their final arrangement to me.

3 Gross color matching

The first important decision is how to represent and compute the color histogram. It was obvious to me that a 3D matrix would be very inefficient due to the large amount of elements that would be zero, once the images have at most $89 \times 60 = 5340$ different colors (in most cases, way less than that) which is much less than $255^3 = 16,581,375$. Therefore I decided to use a hash table, mapping color (r, g, b) to number of times it appears in the image. The second decision was to discard any pixels with all r, g and b values lower than 10 which are very dark pixels considered by me to be background. The third decision was to break the color key values into intervals. That is, instead of holding every single possible r, g, b color integer values in the hash map, I broke each r, g and b into fixed size intervals. For example, for an interval size of 10, the key $(13, 157, 230)$ would become $(10, 150, 230)$. I will explain later on this section how I compute the normalized L1 comparison.

I found that different interval sizes gave me different visual levels of satisfactory results. To compare then with the survey, I counted for some different interval sizes, how many times the participants' answers matched with one of the 3 most similar image colors and with one of the 3 most unlike image colors. Then I divide those values by 3 (the number of participants, thus taking the average) and by the number of pictures (40, thus normalizing the value into the interval $[0, 1]$. The results of my comparisons are show in table 1.

interval size	Alike color match rate	Unalike color match rate
1	0.1750	0.0833
2	0.2083	0.1083
5	0.3000	0.1000
10	0.4250	0.1083
15	0.4417	0.1500
20	0.5250	0.1833
25	0.5167	0.1583
30	0.5833	0.1917
35	0.5750	0.2000
40	0.6500	0.2250
45	0.5917	0.2167
50	0.6333	0.2333
55	0.6000	0.2417
60	0.5667	0.2417

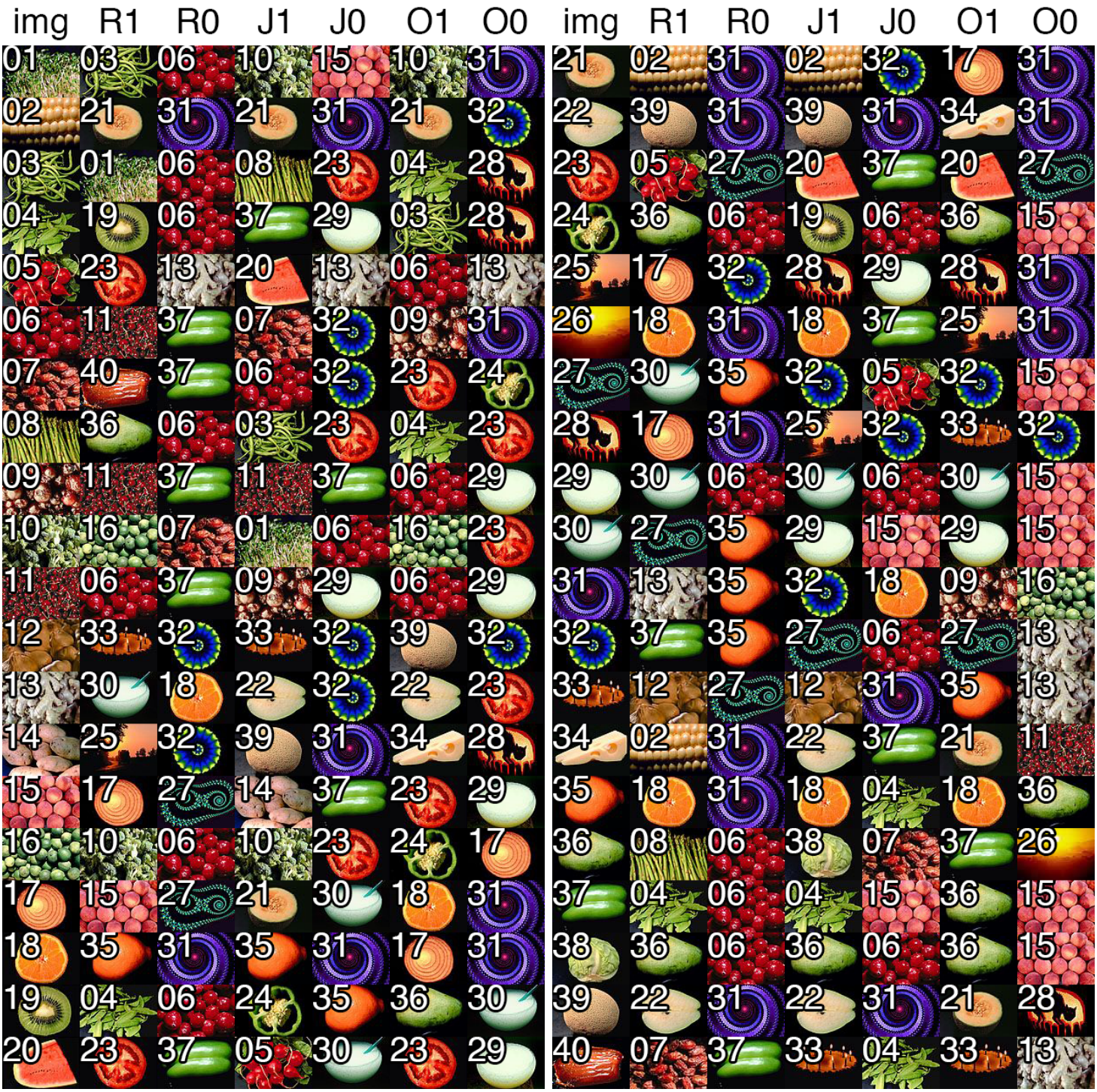
Table 1: Comparison of interval sizes with survey results (gross color matching)

We can see that in most cases, especially for interval size ≥ 20 the alike color match rate is reasonably good, above 50%, while the unalike color match rate is usually much worse, never greater than 25%. In my interpretation that shows that people have a much better notion of how colors are similar than how they are dissimilar. That is, we can easily tell two objects have similar color, but if we see two objects with different colors, although we can certainly tell their colors are different, we can't easily quantise how dissimilar one object is to another. Although it may also be because some people considered objects to have the most different colors in the cases when they have opposite colors (e.g., red and green) instead of considering just the color distribution of the image, like the algorithm did.

In the end I decided to pick the interval size of 40 because the color matching results makes the most sense visually to me and it does very well with the survey comparison. You can see the output of my implementation on figure 6 divided in two halves as a table of 40 rows of septuples just like described in the assignment handout.

Function computeColorHistogram shown below shows how I implemented the histogram computation described above in this section. It receives an array with all images and an integer that holds the number of images and returns and array with one histogram as a hash map for each image.

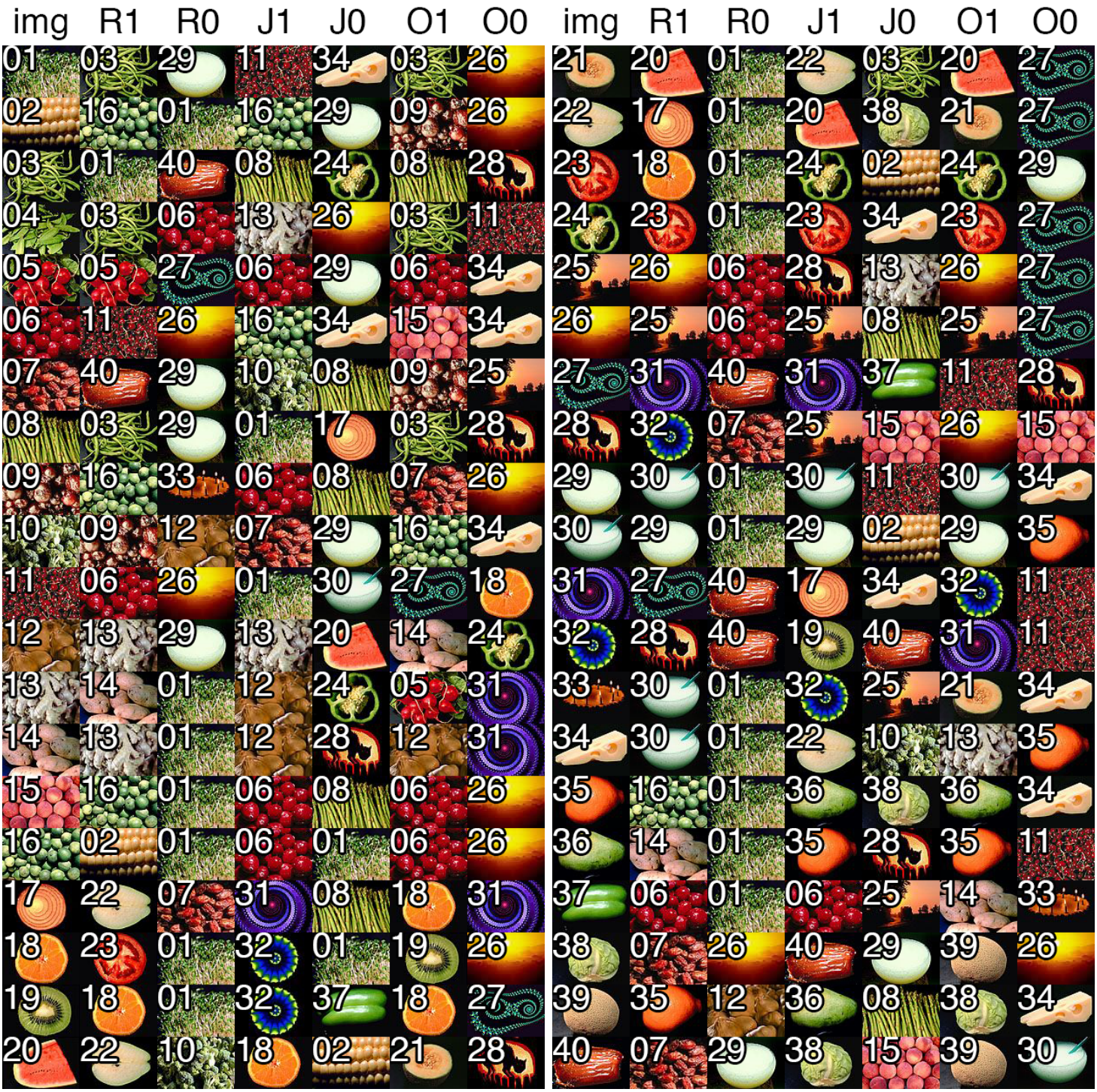
```
function [histograms] = computeColorHistogram( ims, nImages )

    interval_size = 40;
    discard_threshold = 10;
```



(a) Images 1 to 20

(b) Images 21 to 40

Figure 1: Color similarity survey



(a) Images 1 to 20

(b) Images 21 to 40

Figure 2: Texture similarity survey

Figure 3: Rachel's clusters

Figure 4: Joseph's clusters

Figure 5: Oliver's clusters

```
% compute dimensions of images
im = ims{1};
[rows,columns,colors] = size(im);

histograms = {}
for n=1:nImages
        im = ims{n};

        keys={'0,0,0'};
        values={0};
        histogram = containers.Map(keys, values);
```

```
            for i=1:rows
                for j=1:columns
                    key = im(i,j,:);

                    % discard pixels with all r,g and b values lower than 10
                    if ((key(1) < discard_threshold)
                        && (key(2) < discard_threshold)
                        && (key(3) < discard_threshold))
                        continue;
                    end

                    % compute key
                    key = strcat(int2str(key(1)-mod(key(1),interval_size) ),
                        ',',int2str(key(2)-mod(key(2),interval_size) ),
                        ',',int2str(key(3)-mod(key(3),interval_size) ));

                    if(histogram.isKey(key))
                        histogram(key) = 1 + histogram(key);
                    else
                        histogram(key) = 1;
                    end

                end
            end

            remove(histogram,{'0,0,0'});

            histograms{n} = histogram;
        end

end
```

Function colorMatching shown below shows how I compute the normalized L1 comparison. It receives two image histograms as hash maps and the number of rows and columns of the images and returns the normalized L1 comparison as described previously. To compute the normalized L1 comparison between a pair of histogram, I check for each key in the union of keys from both histograms the absolute difference of value between the correspondent values in each histograms, assuming a value of 0 for the histogram that doesn't have the specific key and I sum up all those absolute differences. If the images didn't have any pixels in common and all pixels in the images were different from one another then the images would be the most different possible and the absolute total difference would be biggestDif = number of rows $\times$ number of columns $\times$3(number of color channels) $\times$2 (number of images). Therefore I use this value to normalize the absolute difference, that is, I divide the absolute difference I acquired in the steps above by biggestDif. And as a last step I subtract the current normalized difference from 1 to have 0 represent the least similarity and 1 the most similarity.

```
function [matching] = colorMatching( histogram1, histogram2, rows, columns )
    keys1 = histogram1.keys();
    keys2 = histogram2.keys();

    keys= union(keys1,keys2);

    numberOfKeys = size(keys);
    numberOfKeys = numberOfKeys(2);

    L1 = 0;

    for i=1:numberOfKeys
        key = keys{i};

        val1 = 0;
        val2 = 0;
```

```
            if histogram1.isKey(key)
                    val1 = histogram1(key);
            end

            if histogram2.isKey(key)
                    val2 = histogram2(key);
            end

            dif = abs(val1 - val2);

            L1 = L1 + dif;
        end

        biggestDif = rows*columns*3*2;
        matching = 1-L1/biggestDif;
end
```

In order to explain how I computed the 4 most alike and 4 most unalike images in color, I have to first explain my strategy adopted for computing clusters. MATLAB has built-in functionality for computing hierarchical clustering, which is explained at www.mathworks.com/help/stats/hierarchical-clustering.html. Basically, if you have a $N \times N$ distance matrix Y such that each $Y(i,j) \in [0,1]$ holds the distance of the i^{th} element to the j^{th}, then the following

```
Z=linkage(Y);
```

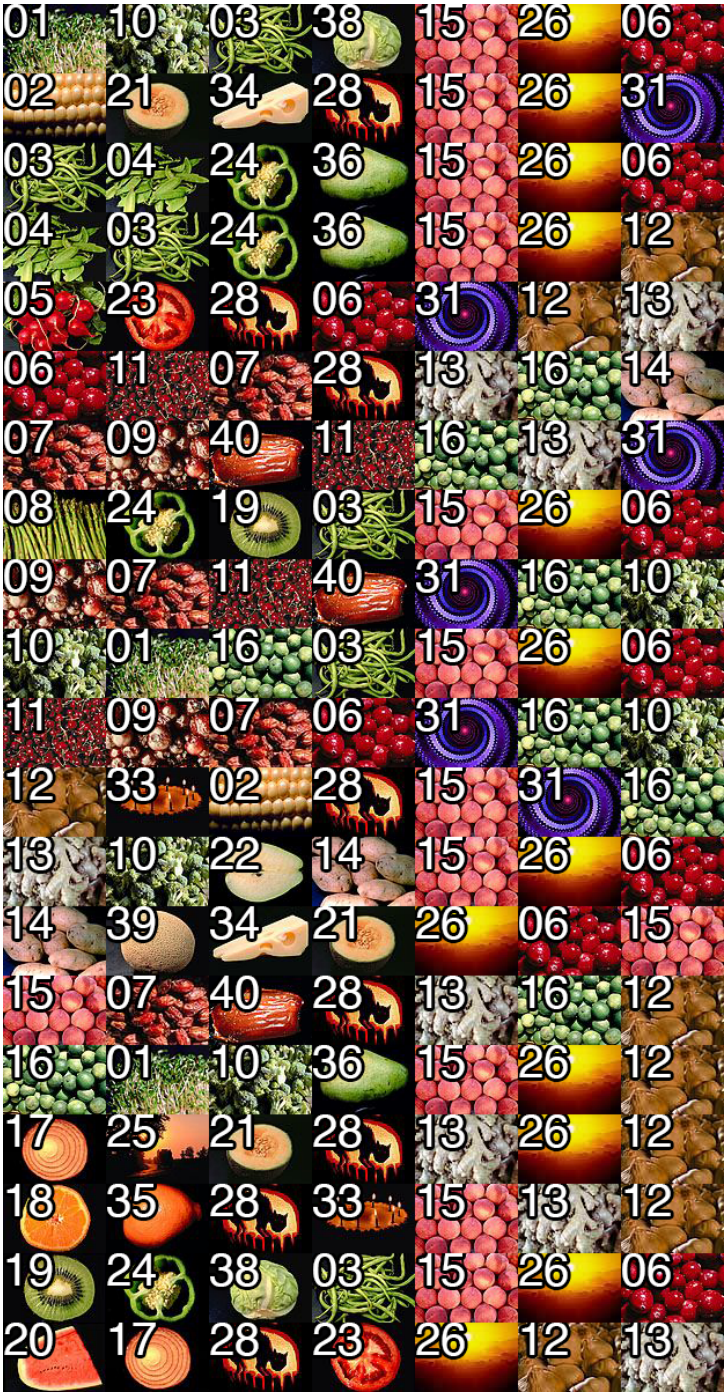
returns the step-by-step combination of elements into clusters as explained in the link above under section "Linkages". By default it does it as complete-link, but a second parameter 'single' may be passed in order to do single-link linkage. In order to actually get the clusters we can do the following, for example:

```
T = cluster(Z,'maxclust', 3);
```

Which clusters the elements into 3 clusters and returns an $N \times 1$ vector such that each element $T(i)$ has the number of the cluster that the i^{th} element belongs to. More details in the link above under section "Create Clusters".

With that explained I can describe how I computed the 4 most alike and 4 most unalike images in color. For the 4 most alike images, my reasoning was to follow the step-by-step of the complete-link clustering, and the first time a cluster with 4 images or more is formed, we have the 4 most similar images. Because complete-link combines clusters that are closer first and it makes clusters such that all images are related, this seemed like a very good strategy to me. Function fourMostalike shown below receives the difs $N \times N$ matrix with the normalized L1 difference of all images and integer nImages with the number of images and returns vector most_alike with the 4 most alike images. Notice that it is possible that there will be a step in which two clusters of size 3 will be combined into a cluster of size 6. In this case we return just the first 4 images of this cluster.

```
function [most_alike] = fourMostAlike( difs, nImages )

        clusters_closeness = 1-difs;

        % find first four images to stay together
        complete_linkage = linkage(clusters_closeness, 'complete');

        clusters = {}
        for i=1:nImages
                clusters{i} =[i];
        end

        most_alike = []
        for i=1:(nImages-1)
                new_i = length(clusters)+1
                clusters{new_i} =
                        [clusters{complete_linkage(i,1)} clusters{complete_linkage(i,2)}]
                if (length(clusters{new_i}) >= 4)
                        most_alike = clusters{new_i};
                        break;
```



(a) Images 1 to 20

(b) Images 21 to 40

Figure 6: Gross color matching

```
        end
    end

    most_alike = most_alike(1:4);

end
```

For the 4 most unalike images, my reasoning was to follow the step-by-step of the single-link clustering, and the first time we have only 4 images that are in single-element clusters, then the four most unalike images are those 4. Because single-link clustering always combines two clusters that have at least one image in each of them that are related, then the last 4 images to be combined have to be very different from all the others in order to have not joined a cluster by that point. Function fourMostUnalike shown below similarly receives the difs $N \times N$ matrix with the normalized L1 difference of all images and integer nImages with the number of images and returns vector most_unalike with the 4 most unalike images. Notice that in some step we might go from 5 single clusters to 3. In this case we arbitrarily pick the image with smallest index to join the other 3.

```
function [most_unalike] = fourMostUnalike( difs, nImages )

    clusters_closeness = 1-difs;

    % find first four images to be left alone
    single_linkage = linkage(clusters_closeness, 'single');

    clusters = {}
    for i=1:nImages
        clusters{i} =[i];
    end

    n=nImages;

    for i=1:(nImages-1)
        el1 = single_linkage(i,1)
        el2 = single_linkage(i,2)

        if(el1 <= nImages)
            n = n -1;
            clusters{el1} = [];
        end

        if (n == 4)
            break;
        end

        if(el2 <= nImages)
            n = n -1;
            clusters{el2} = [];
        end

        if (n == 4)
            break;
        end

    end

    % put the result in a vector
    most_unalike = [];
    n=1;
    for i=1:nImages
        if (isempty(clusters{i}))
            continue;
        end
```

```
            most_unalike(n) = clusters{i};
            n = n+1;
        end

end
```

You can see the output of my implementation on figure 7.

Figure 7: Four most alike images and four most unalike images in color

Wrapping it all up, we have the main function part1 for the first part of this assignment shown below. It first loads all images and labeled images (with numbers on them), then it calls the function that computes all the histograms. Next, it computes the difs matrix, with all pair-wise normalized L1 differences by calling function colorMatching for all pairs of histograms. Next we save the results as images. Then we compute the four most alike and four most unalike images by calling fourMostAlike and fourMostUnalike functions and then we display and save the results. The function returns the difs matrix, which can be used to compare with the survey results.

```
function [difs] = part1( )

        % number of images
        nImages = 40;

        % load images
        ims = {}
        labeled_ims = {}
        for i=1:nImages
                if i < 10
                        ims{i} = imread(strcat('imgs/i0',int2str(i),'.ppm'));
                        labeled_ims{i} = imread(strcat('labeledimgs/i0',int2str(i),'.png'));
                else
                        ims{i} = imread(strcat('imgs/i',int2str(i),'.ppm'));
                        labeled_ims{i} = imread(strcat('labeledimgs/i',int2str(i),'.png'));
                end
        end

        % compute dimensions of images
        im = ims{1};
        [rows,columns,colors] = size(im);

        % compute all histograms
        histograms = computeColorHistogram(ims, nImages);
```

```
% compute the difs matrix (similarities)
difs = zeros(nImages,nImages);
for i=1:nImages
        for j=i:nImages
                tic;
                if (i==j)
                        difs(i,j) = 1;
                        continue;
                else
                        dif = colorMatching( histograms{i}, histograms{j},
                                rows, columns);
                        difs(i,j) = dif;
                        difs(j,i) = dif;
                end
                toc;
                [i,j]
        end
end

% sort by similarity
[difs_sorted difs_index] = sort(difs);

% keep only 3 most similar and 3 most dissimilar
difs_index = [difs_index(1:3,:); difs_index(nImages-2-1:nImages-1,:)];

% generate image table with results and save it in two halves
result_image = []
for j=1:(nImages/2)
        result_row = [labeled_ims{j} labeled_ims{difs_index(6,j)}
                labeled_ims{difs_index(5,j)} labeled_ims{difs_index(4,j)}
                labeled_ims{difs_index(1,j)} labeled_ims{difs_index(2,j)}
                labeled_ims{difs_index(3,j)}];
        if j==1
                result_image = result_row;
        else
                result_image = [result_image; result_row];
        end
end
imwrite(result_image,'part1/part1_1.png');
result_image = []
for j=(nImages/2+1):nImages
        result_row = [labeled_ims{j} labeled_ims{difs_index(6,j)}
                labeled_ims{difs_index(5,j)} labeled_ims{difs_index(4,j)}
                labeled_ims{difs_index(1,j)} labeled_ims{difs_index(2,j)}
                labeled_ims{difs_index(3,j)}];
        if j==1
                result_image = result_row;
        else
                result_image = [result_image; result_row];
        end
end
imwrite(result_image,'part1/part1_2.png');

% compute 4 most alike and 4 most unalike images
most_alike = fourMostAlike( difs, nImages );
most_unalike = fourMostUnalike( difs, nImages );
% Display and save them
alike_imgs = []
unalike_imgs = []
for i=1:4
```

```
        alike_imgs = [alike_imgs labeled_ims{most_alike(i)}];
        unalike_imgs = [unalike_imgs labeled_ims{most_unalike(i)}];
    end
    f=figure
    subplot(2,1,1);
    imshow(alike_imgs);
    title('Four most alike images in color');
    subplot(2,1,2);
    imshow(unalike_imgs);
    title('Four most unalike images in color');
    saveas(f,'part1/fourmost_color.png');

end
```

4 Gross texture matching

The first thing to do here is computing the Laplacian of the images. For that, I implemented the function imgsAndLaplacians that receives the number of images as parameter and loads all the images, labeled images and compute all the image Laplacians just like described in the assignment handout. First it turns the image into grayscale by taking the average of the r, g and b values and then compute the Laplacian by making each pixel 8 times its value subtracting the values of its 8 neighbors.

```
function [ims, laplacians, labeled_ims] = imgsAndLaplacians(nImages)

    laplacians = {}
    ims = {}
    labeled_ims = {}

    for k=1:nImages

        if k < 10
            im = imread(strcat('imgs/i0',int2str(k),'.ppm'));
            labeled_ims{k} = imread(strcat('labeledimgs/i0',int2str(k),'.png'));
        else
            im = imread(strcat('imgs/i',int2str(k),'.ppm'));
            labeled_ims{k} = imread(strcat('labeledimgs/i',int2str(k),'.png'));
        end

        ims{k} = im;

        % Convert to grayscale (R + G + B)/3
        grayIm = (im(:,:,1) + im(:,:,2) + im(:,:,2))/3;

        % Laplacian
        [rows,columns] = size(grayIm);
        % make it double type to allow for negative values
        laplacian = double(grayIm)*8;

        for i=1:rows
            for j=1:columns
                for a = max(1,i-1) : min(rows,i+1)
                    for b=max(1,j-1):min(columns,j+1)
                        if(a ~=i || b ~= j)
                            laplacian(i,j) =
                                laplacian(i,j)
                                - double(grayIm(a,b));
                        end
                    end
                end
            end
```

```
            end

            laplacians{k} = laplacian;
        end

end
```

Once I have computed the Laplacians, I compute the histograms of the Laplacians exactly like I did for the images in the gross color matching step, except that the key constitutes of just one value, instead of a 3-tuple and also I don't delete dark pixels. The function computeTextureHistogram can be seen below:

```
function [histograms] = computeTextureHistogram( laplacians, nImages )

        divide_space_by = 30;

        % compute dimensions of images
        lapIm = laplacians{1};
        [rows,columns] = size(lapIm);

        histograms = {}
        for n=1:nImages
                lapIm = laplacians{n};

                keys={'0'};
                values={0};
                histogram = containers.Map(keys, values);

                for i=1:rows
                        for j=1:columns

                                % compute key
                                key = lapIm(i,j);
                                key = int2str(key-mod(key,divide_space_by));

                                if(histogram.isKey(key))
                                        histogram(key) = 1 + histogram(key);
                                else
                                        histogram(key) = 1;
                                end
                        end
                end

                histograms{n} = histogram;
        end

end
```

Like in the previous section, here I also experiment with different values for the interval size and the results are shown in table 2. The match rate for texture was significantly lower than for color. I believe that's because texture is a much harder feature for people to compare than color is. People have to "look harder" to compare texture and a lot of times it is just too hard to for them tell what the most similar is.

I picked 30 for interval size because the texture matching results makes the most sense and it does very well with the survey comparison. You can see the output of my implementation on figure 8 divided in two halves as a table of 40 rows of septuples just like described in the assignment handout.

Function textureMatching shown below computes the normalized L1 comparison exactly like colorMatching except that it uses a different value for biggestDif, which is number of rows $\times$ number of columns $\times$2(number of images).

```
function [matching] = textureMatching( histogram1, histogram2, rows, columns )
        keys1 = histogram1.keys();
        keys2 = histogram2.keys();
```

(a) Images 1 to 20

(b) Images 21 to 40

Figure 8: Gross texture matching

interval size	Alike texture match rate	Unalike texture match rate
1	0.1833	0.1250
2	0.2083	0.1333
5	0.2083	0.1333
10	0.2417	0.1417
15	0.2417	0.1500
20	0.2667	0.1417
25	0.2583	0.1500
30	0.2833	0.1583
35	0.2583	0.1667
40	0.2750	0.1417
45	0.2333	0.1500
50	0.2333	0.1500
55	0.2417	0.1583
60	0.2500	0.1583

Table 2: Comparison of interval sizes with survey results (gross texture matching)

```
        keys= union(keys1,keys2);

        numberOfKeys = size(keys);
        numberOfKeys = numberOfKeys(2);

        L1 = 0;

        for i=1:numberOfKeys
                key = keys{i};

                val1 = 0;
                val2 = 0;

                if histogram1.isKey(key)
                        val1 = histogram1(key);
                end

                if histogram2.isKey(key)
                        val2 = histogram2(key);
                end

                dif = abs(val1 - val2);

                L1 = L1 + dif;
        end

        biggestDif = rows*columns*2;
        matching = 1-L1/biggestDif;
end
```

For computing the four most alike images and the four most unalike images, I did exactly what I did in the gross color matching step and the results are show in figure 9.

Wrapping it all up, we have the main function *part2* for the first part of this assignment shown below. It first loads all images, labeled images and Laplacians, then it works analogously to part1.

```
function [difs] = part2( )

        % number of images
        nImages = 40;
```

Figure 9: Four most alike images and four most unalike images in texture

```
% load images and compute laplacians
[ims, laplacians, labeled_ims] = imgsAndLaplacians(nImages);

% compute dimensions of images
im = ims{1};
[rows, columns, colors] = size(im);

% compute all histograms
histograms = computeTextureHistogram(laplacians, nImages);

% compute the difs matrix (similarities)
difs = zeros(nImages,nImages);
for i=1:nImages
        for j=i:nImages
                if (i==j)
                        difs(i,j) = 1;
                        continue;
                else
                        dif = textureMatching( histograms{i}, histograms{j},
                                rows, columns);
                        difs(i,j) = dif;
                        difs(j,i) = dif;
                end
        end
end

% sort by similarity
[difs_sorted difs_index] = sort(difs);

% keep only 3 most similar and 3 most dissimilar
difs_index = [difs_index(1:3,:); difs_index(nImages-2-1:nImages-1,:)];

% generate image table with results and save it in two halves
result_image = []
for j=1:(nImages/2)
        result_row = [labeled_ims{j} labeled_ims{difs_index(6,j)}
                labeled_ims{difs_index(5,j)} labeled_ims{difs_index(4,j)}
                labeled_ims{difs_index(1,j)} labeled_ims{difs_index(2,j)}
```

```
                labeled_ims{difs_index(3,j)}];
        if j==1
                result_image = result_row;
        else
                result_image = [result_image; result_row];
        end
end
imwrite(result_image,'part2/part2_1.png');
result_image = []
for j=(nImages/2+1):nImages
        result_row = [labeled_ims{j} labeled_ims{difs_index(6,j)}
                labeled_ims{difs_index(5,j)} labeled_ims{difs_index(4,j)}
                labeled_ims{difs_index(1,j)} labeled_ims{difs_index(2,j)}
                labeled_ims{difs_index(3,j)}];
        if j==1
                result_image = result_row;
        else
                result_image = [result_image; result_row];
        end
end
imwrite(result_image,'part2/part2_2.png');

% compute 4 most alike and 4 most unalike images
most_alike = fourMostAlike( difs, nImages );
most_unalike = fourMostUnalike( difs, nImages );
% Display and save them
alike_imgs = []
unalike_imgs = []
for i=1:4
        alike_imgs = [alike_imgs labeled_ims{most_alike(i)}];
        unalike_imgs = [unalike_imgs labeled_ims{most_unalike(i)}];
end
f=figure
subplot(2,1,1);
imshow(alike_imgs);
title('Four most alike images in texture');
subplot(2,1,2);
imshow(unalike_imgs);
title('Four most unalike images in texture');

saveas(f,'part2/fourmost_texture.png');

end
```

5 Combine similarities, and cluster

In this section I will show the final code first and then explain all the steps.

```
function [complete_clusters, single_clusters] = part3( )

        % number of images
        nImages = 40;

        % load images and compute laplacians
        [ims, laplacians, labeled_ims] = imgsAndLaplacians(nImages);

        % compute dimensions of images
        im = ims{1};
        [rows,columns,colors] = size(im);
```

```
% compute all histograms
colorHistograms = computeColorHistogram(ims, nImages);
textureHistograms = computeTextureHistogram(laplacians, nImages);

% compute the difs matrix (similarities)
difs = zeros(nImages,nImages);
for i=1:nImages
        for j=i:nImages
                if (i==j)
                        difs(i,j) = 1;
                        continue;
                else
                        dif = computeSimilarity( colorHistograms{i},
                                colorHistograms{j}, textureHistograms{i},
                                textureHistograms{j}, rows, columns);

                        difs(i,j) = dif;
                        difs(j,i) = dif;
                end
        end
end

clusters_closeness = 1-difs;

% number of clusters
k=7;

% compute clusters
complete_linkage = linkage(clusters_closeness, 'complete');
single_linkage = linkage(clusters_closeness, 'single');
complete_clusters = cluster(complete_linkage, 'maxclust', k);
single_clusters = cluster(single_linkage, 'maxclust', k);

% save cluster images
complete_clusters_array = {}
single_clusters_array = {}
for i=1:k
        complete_clusters_array{i} = []
        single_clusters_array{i} = []
end
for i=1:nImages
        complete_clusters_array{complete_clusters(i)}
                = [complete_clusters_array{complete_clusters(i)} i];
        single_clusters_array{single_clusters(i)}
                = [single_clusters_array{single_clusters(i)} i];
end
complete_cluster_imgs = clustersImage(complete_clusters_array, labeled_ims);
single_cluster_imgs = clustersImage(single_clusters_array, labeled_ims);
imwrite(complete_cluster_imgs,'part3/complete_cluster_imgs.png');
imwrite(single_cluster_imgs,'part3/single_cluster_imgs.png');

% defining color threshold in order for each one of the
% seven clusters have their own unique color
complete_threshold = sort(complete_linkage(:,3));
complete_threshold = complete_threshold(size(complete_linkage,1)+2-k);
single_threshold = sort(single_linkage(:,3));
single_threshold = single_threshold(size(single_linkage,1)+2-k);

% show and save dendograms
```

```
f=figure();
set(f,'Visible','off');
dendrogram(complete_linkage,0,'ColorThreshold',complete_threshold);
title('Complete Link');
set(f, 'PaperUnits', 'centimeters');
set(f, 'PaperPosition', [0 0 30 25]);
saveas(f,'part3/complete_link_dendrogram.png')
f=figure();
set(f,'Visible','off');
dendrogram(single_linkage,0,'ColorThreshold',single_threshold);
title('Single Link');
set(f, 'PaperUnits', 'centimeters');
set(f, 'PaperPosition', [0 0 30 25]);
saveas(f,'part3/single_link_dendrogram.png')
```

end

As you can see, we first compute all the color and texture histograms and then we compute the difs matrix based on the computeSimilarity function shown below that combines the similarities. computeSimilarity takes the color and texture histograms of the two images to be compared, the number of rows and columns in the images and returns the similarity value in the interval $[0, 1]$.

```
function [S] = computeSimilarity( colorHistogram1, colorHistogram2,
    textureHistogram1, textureHistogram2, rows, columns )

    r = 0.2;
    C = colorMatching( colorHistogram1, colorHistogram2, rows, columns);
    T = textureMatching( textureHistogram1, textureHistogram2, rows, columns );
    S = r*T + (1-r)*C;
```

end

After computing the difs matrix, we are ready to compute the clusters. That can be easily done using the functions linkage and cluster as explained in section 3. After acquiring the clusters, we turn then into array of image indexes and pass them to clustersImage, which computes the image with all the cluters of images shown in different rows.

```
function [all_cluster_imgs] = clustersImage( clusters_array, ims )

    img_size = size(ims{1});

    max_cluster_size = 1;
    all_cluster_imgs = [];
    for i=1:7
        cluster_size = length(clusters_array{i});
        cluster = clusters_array{i};
        cluster_imgs = [];

        for j=1:cluster_size
            im = ims{cluster(j)};
            cluster_imgs = [cluster_imgs im];
        end

        if(i==1)
            all_cluster_imgs = cluster_imgs;
            max_cluster_size = cluster_size;
            continue;
        end

        if (cluster_size > max_cluster_size)
            horizontal_rectangles = cluster_size - max_cluster_size;
            padding = zeros([size(all_cluster_imgs,1)
```

```
                    img_size(2)*horizontal_rectangles 3]);
                all_cluster_imgs = [all_cluster_imgs padding];
                max_cluster_size = cluster_size;
            elseif (cluster_size < max_cluster_size)
                horizontal_rectangles = max_cluster_size - cluster_size;
                padding = zeros([img_size(1) img_size(2)*horizontal_rectangles 3]);
                cluster_imgs = [cluster_imgs padding];
            end
            all_cluster_imgs = [all_cluster_imgs; cluster_imgs];
        end

end
```

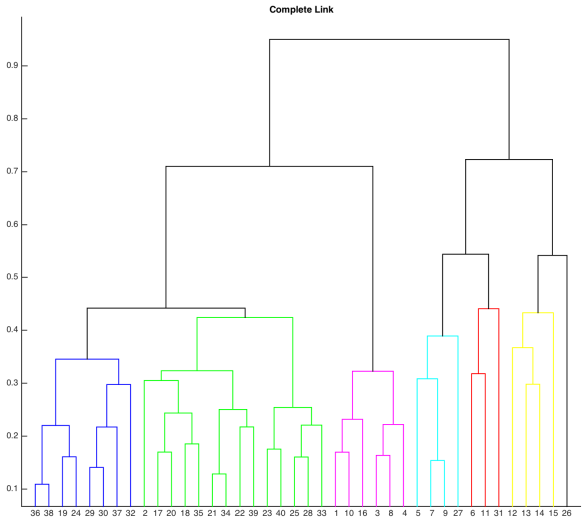
We also display some dendograms, which are explained in the link www.mathworks.com/help/stats/hierarchical-clustering.html under "Dendograms". We pass a 0 argument to it in order for it not to collapse the data points to 30 and define a color threshold in order for each one of the seven clusters have their own unique color.

In this section the parameter that I had to experiment with was value r in the function computeSimilarity. Each value would give me different complete-link and single-link clusters and I compared those to the ones created by the survey participants. In order to compare the different clusters I had to determine how to measure the similarity between two partitions of a set. I did so by implementing the Rand Index as explained at en.wikipedia.org/wiki/Rand_index.

My implementation can be seen below in function randIndex. It receives two clusters represented as arrays such that each element contains the index of the cluster that that element belongs to (it is the same kind of array returned by function cluster), the number of elements n and then it returns the rand index.

```
function [index] = randIndex( P1, P2, n )

    % computing the rand index of two set partitions based on
    % http://en.wikipedia.org/wiki/Rand_index

    a = 0;
    b = 0;
    c = 0;
    d = 0;

    for i=1:n
        for j=i:n

            if (i==j)
                continue;
            end

            P1_same_partition_ij = (P1(i) == P1(j));
            P2_same_partition_ij = (P2(i) == P2(j));

            if (P1_same_partition_ij && P2_same_partition_ij)
                a = a +1;
        elseif (~P1_same_partition_ij && ~P2_same_partition_ij)
                b = b +1;
        elseif (P1_same_partition_ij && ~P2_same_partition_ij)
                c = c + 1;
            else
                d = d + 1;
        end
    end

    index = (a+b) / (a + b + c + d);

end
```

(a) Complete-link cluster with $r = 0.2$



(b) Complete-link dendogram with $r = 0.2$

Figure 10: Complete-link with $r = 0.2$

The results of my comparisons are shown in table 3. The first column has all the values of r tested, the second column has the value of the average of the rand index between the complete-link cluster generated with all 3 participants' clusters and the third column similarly for the single-link cluster. The values of r that favor more color than texture (< 0.5) seems to match more with the participants' responses. I believe this is because people tend to naturally read color before texture as measure of similarity.

r	complete link average rand index	single link average rand index
0.0	0.7748	0.3581
0.1	0.8030	0.4778
0.2	0.7983	0.6560
0.3	0.7735	0.6410
0.4	0.7628	0.6115
0.5	0.7735	0.6115
0.6	0.7397	0.5927
0.7	0.7291	0.5927
0.8	0.7291	0.5744
0.9	0.7530	0.5910
1.0	0.7530	0.5910

Table 3: Rand index cluster comparison

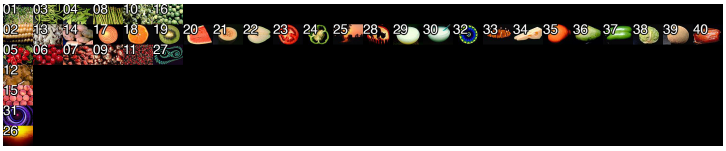
My pick for r was 0.2, because I felt like it was the most well balanced and I feel like color is a stronger factor for humans' perceived similarity. You can see the output of the program with $r = 0.2$ in images 10 and 11.

In figure 12, you can see the output for $r = 0.1$ for complete-link, which had the best average rand index compared to the participants' responses.

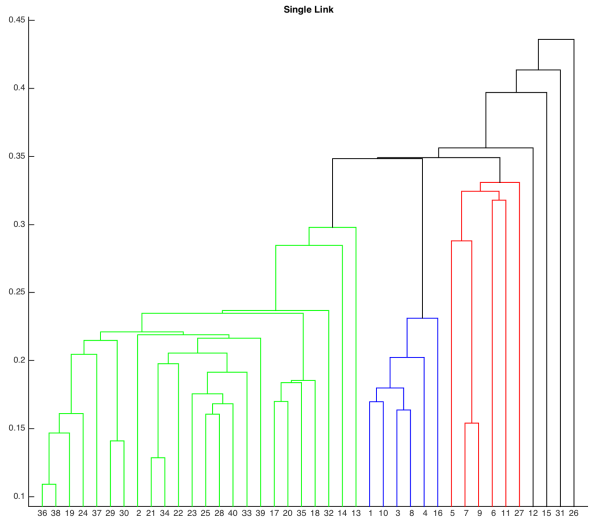
In figures 13 and 14, you can see the output for $r = 0.0$, just taking into consideration color. Notice that the single-link cluster in this case had the worst comparison with the participant's answers (0.3581), once it created 6 groups with very dissimilar images and one large group with all the rest.

Figures 15 and 16 show the output for $r = 0.5$, taking into consideration half color and half textures.

Figures 17 and 18 show the output for $r = 1.0$, just taking textures into consideration.

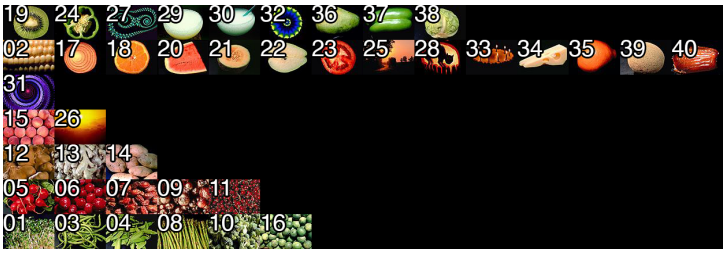


(a) Single-link cluster with $r = 0.2$

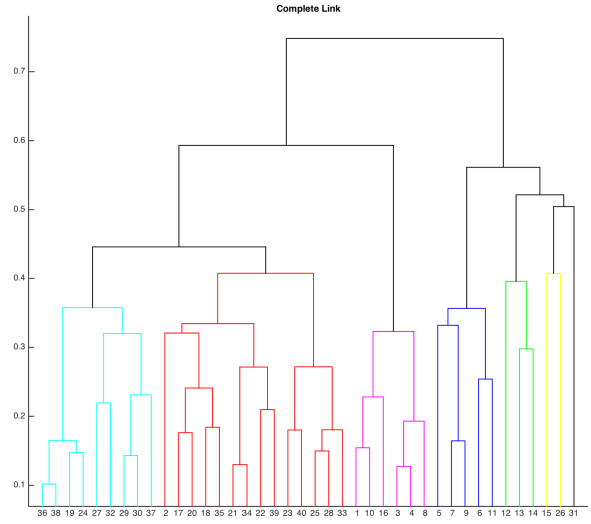


(b) Single-link dendogram with $r = 0.2$

Figure 11: Single-link with $r = 0.2$

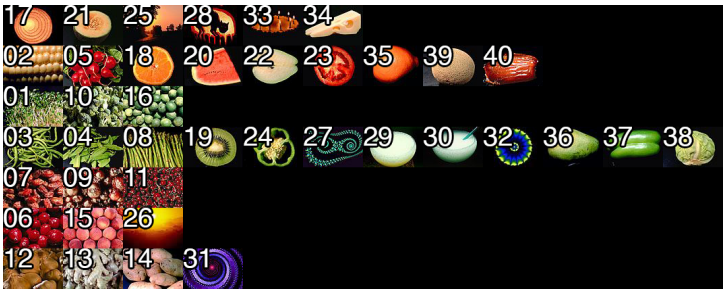


(a) Complete-link cluster with $r = 0.1$

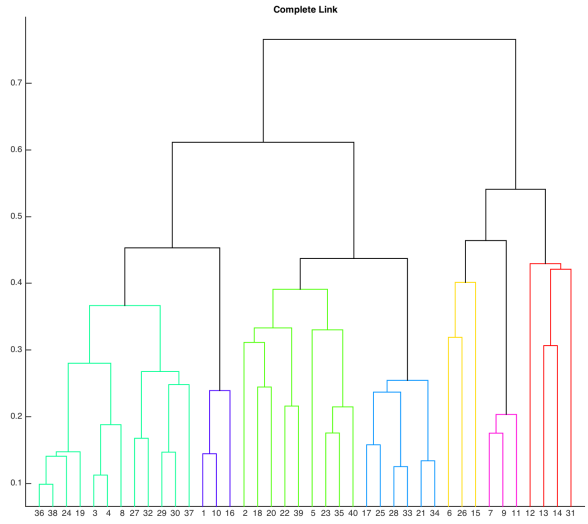


(b) Complete-link dendogram with $r = 0.1$

Figure 12: Complete-link with $r = 0.1$



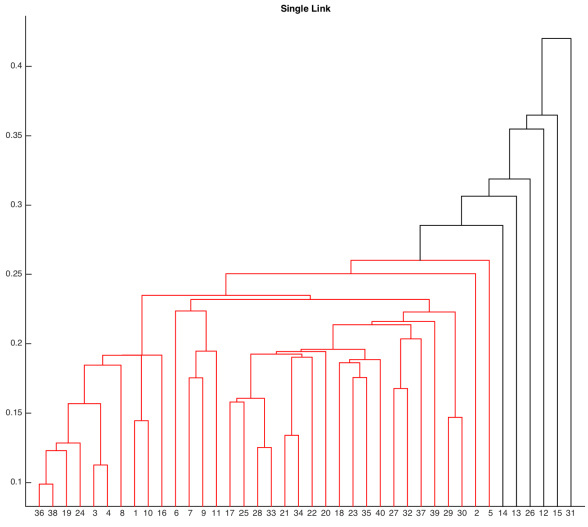
(a) Complete-link cluster with $r = 0.0$



(b) Complete-link dendogram with $r = 0.0$

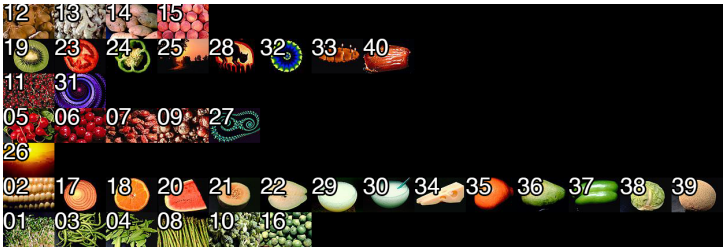
Figure 13: Complete-link with $r = 0.0$

(a) Single-link cluster with $r = 0.0$

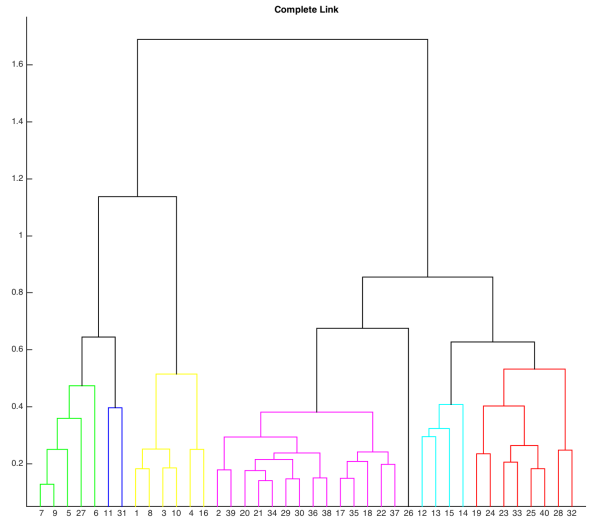


(b) Single-link dendogram with $r = 0.0$

Figure 14: Single-link with $r = 0.0$

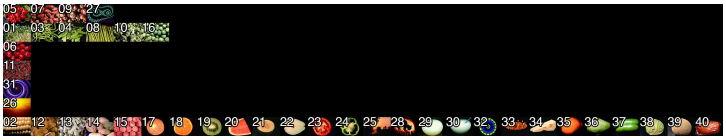


(a) Complete-link cluster with $r = 0.5$

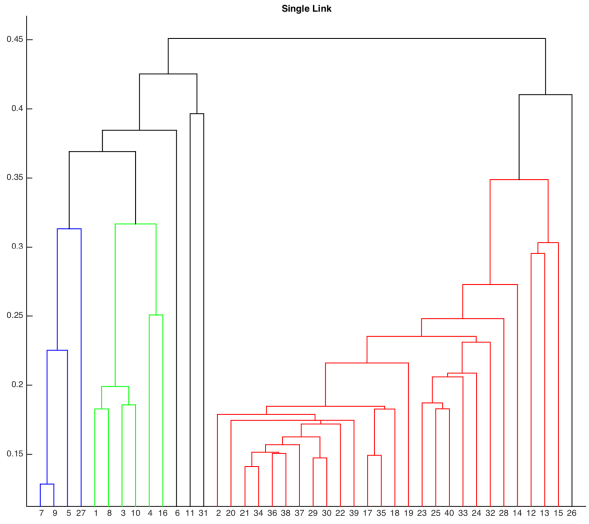


(b) Complete-link dendogram with $r = 0.5$

Figure 15: Complete-link with $r = 0.5$



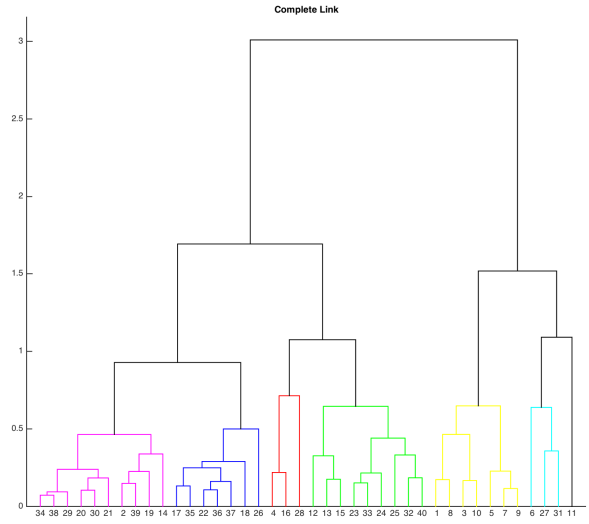
(a) Single-link cluster with $r = 0.5$



(b) Single-link dendogram with $r = 0.5$

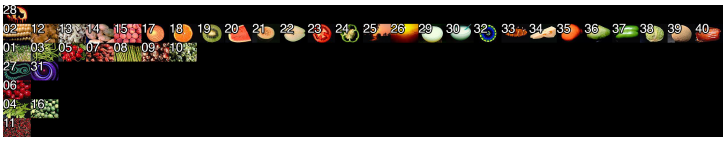
Figure 16: Single-link with $r = 0.5$

(a) Complete-link cluster with $r = 1.0$

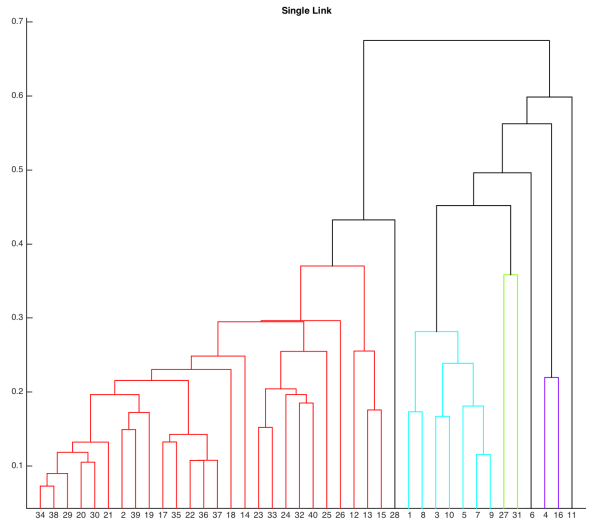


(b) Complete-link dendogram with $r = 1.0$

Figure 17: Complete-link with $r = 1.0$



(a) Single-link cluster with $r = 1.0$



(b) Single-link dendogram with $r = 1.0$

Figure 18: Single-link with $r = 1.0$

6 Appendix: Accessing the Data

Included in my submission is a workspace file called "surveyworkspace.mat" with all the data collected in the survey. The variables R1, R2, R3, J1, J2, J3, O1, O2 and O3 are respectively Rachel's answers to task 1, 2 and 3 and similarly for Joseph and Oliver. The answers for task 1 and 2 are 40×2 matrices with the answer for most similar and most dissimilar image in each row. The answers for task 3 is a vector that represent the clusters in the same format as the one returned by function cluster. Each of surveys1, surveys2 and surveys3 are each such that

```
surveys1 = {R1, J1, O1};
surveys2 = {R2, J2, O2};
surveys3 = {R3, J3, O3};
```

The function used to compute the comparisons from sections 3 and 4 is countMatches shown below. It receives the difs matrix returned by either part1() or part2(), an array of survey results (e.g., surveys1) and returns the average normalized rate of match between the survey results and the algorithm's output both for the most alike images and for the most unalike images.

```
function [matches] = countMatches( difs, surveys, nImages )

        % sort by similarity
        [difs_sorted difs_index] = sort(difs);
        difs_index = [difs_index(1:3,:); difs_index(nImages-2-1:nImages-1,:)]

        % keep only 3 most similar and 3 most dissimilar
        difs_sorted = [difs_sorted(1:3,:); difs_sorted(nImages-2-1:nImages-1,:)]
        % transpose the matrices
        difs_index = difs_index'
        difs_sorted = difs_sorted'

        % rearrange so each row has the 3 most
        % similar images followed by the 3 least similar
        difs_sorted = [difs_sorted(:,6) difs_sorted(:,5) difs_sorted(:,4) difs_sorted(:,1:3)]
        difs_index = [difs_index(:,6) difs_index(:,5) difs_index(:,4) difs_index(:,1:3)]

        matches = [0 0 0 0 0 0];

        % count time survey answers matches with algorithm's
        n = length(surveys);
        for s_i=1:n
                survey = surveys{s_i};
                for i=1:nImages
                        for j=1:6
                                if (difs_index(i,j) == survey(i,floor((j-1)/3)+1))
                                        matches(j) = matches(j) + 1;
                                end
                        end
                end
        end

        % sum matches
        matches = [sum(matches(1:3)) sum(matches(4:6))];
        % normalize
        matches = matches/(nImages*n);

end
```

You can perform a comparison between data and the algorithms the following way:

```
difs = part1();
countMatches(difs,surveys1, 40)
```

```
ans =
    0.6500    0.2250

difs = part2();
countMatches(difs,surveys2, 40)

ans =
    0.2833    0.1583

[complete_clusters, single_clusters] = part3( );

randIndex(complete_clusters, J3, 40)

ans =
    0.7410

randIndex(single_clusters, R3, 40)

ans =
    0.7295
```