Problem Set 2 – K-Means Clustering and Vector Quantization
CMSC 426.
Due Feb. 19

Programming Assignment

For this assignment, you will implement the k-means clustering algorithm and use it to perform vector quantization on a color image. You can test the various stages of your program using the function ps2, which is available on the class web page. I will show you the results of my program for each part of the problem. However, there are some choices to be made in how you represent the intermediate results of your program. You can do it as I did, or in a different way, as long as you document your choices. You are also free to change the number and meaning of the arguments of each function, as long as you document your choices. For each problem, your writeup should mention any changes you make to the representation, but can be minimal if there isn't much to say.

1. **15 Points:** Implement a function, `D = cluster_points_distance(cs, ps),` to compute the squared distance between a set of points and a set of cluster centers. For color quantization, each "point" will represent the color at a single pixel. So it will be a 3D point, holding RGB values. The cluster centers are also a set of 3D points. Of course, there is nothing in the process of computing distance that depends on the points representing colors.

   I choose to represent the points, ps, using a 3xn matrix, in which each column holds the coordinates of a single 3D point. Similarly, cs is a 3xk matrix representing cluster centers. This is illustrated at the top of ps2, where I assign simple test values to the variables cs and ps. I find this convenient, but you could do something different. For example, you could represent points using a 3D matrix that is just like a color image. My way, you'll have to convert a color image to my representation, but my representation is simpler in some ways too.

   My function cluster_points_distance then returns the distances in a kxn matrix, D. So D(i,j) holds the squared distance between cluster center i, which is in cs(:,i), and point j, which is in ps(:,j).

   Running my program looks like:

```
>> ps2(1)

D =

     6    22   105
```

```
    17     3    26

>>
```

2. **15 points:** Next, implement `mems = cluster_members(cs, ps).` This function figures out which cluster each point belongs to. That is, point one belongs to cluster one if it is closer to that cluster center than to any other. I choose to represent the result with a vector, mems, which has length n. Every element of mems is a number between 1 and k. So if mems(i) = j, this means that point i belongs to cluster j. Again, you are free to represent cluster membership in a different way, but please explain any different choices you make in your code and writeup. Running my program produces the result:

```
>> ps2(2)

mems =

     1     2     2

>>
```

3. **15 points:** Next, implement `csnew = update_centers(ps, mems).` This creates a new set of cluster centers, given a set of points and their assignment to different clusters. Each new cluster center should be the average of the points assigned to that cluster. `If you find it convenient, you might add a third argument that tells you the number of clusters. So, for example, csnew(:,1) should be the average of all the points that belong to the first cluster (ie., for which mems has a value of 1). Running my code produces:`

```
>> ps2(3)


csnew =

    1.0000    5.0000
    2.0000    3.5000
    1.0000    4.0000

>>
```

4. **25 points:** Finally, put these together to implement: `[cs_final, mems_final] = k_means_initialized(cs, ps, maxIterations).` Here, cs represents some initial set of cluster centers. ps represents the points that we want to cluster. maxIterations is a maximum number of iterations that the program should perform. In principal, we can keep running k-means until it converges, that is, until the assignment of points to clusters does not change. But for color images, it

may take a large number of iterations to converge, so you may want to stop it before it converges. cs_final and mems_final will contain the final set of cluster centers and the assignment of points to cluster centers. Running my program produces:

```
>> ps2(4)

cs_final =

    1.5000    8.0000
    2.5000    4.0000
    2.0000    5.0000


mems_final =

     1     1     2

>>
```

5. **30 points:** Now you can use k-means to quantize an image. That is, given a color image, first select an initial set of cluster centers, and run k-means to select cluster centers and assign each pixel to a cluster. Then create a new image in which each pixel is replaced by the color of the cluster center that it belongs to. This function might have the form: `J = quantize_image(I, k);` Here I is the input image, k is the number of clusters, and J is the final image. So J should have only k different colors in it. The web site shows the image that I produce running ps2(5). This shows a result on the peppers image. In this experiment, I use a maximum of 200 iterations in K-means.

   For your write-up, run your quantize_image program on forest2.jpg (this is the same image we used in Problem Set 1). Generate four images that result from using 8, 16, 32, or 64 cluster centers. Include these images in your writeup, and briefly describe what you have learned about the number of clusters needed for good color quantization. What sorts of errors occur as you have too few clusters? What errors do you still see, even using 64 clusters? Do you have any ideas for fixing these problems, besides just using a very large number of clusters? Also, describe your design choices in implementing this.

   Hint: Remember that images are uint8 (that is, matrices with 8 bits for each number). Performing mathematical operations on these can produce some weird errors, so it might be a good idea to convert these to double (help double) and then back to uint8 when you are done (help uint8).

   Hint: If your program is slow, which it might be if you use a lot of loops, or are running remotely on GLUE, you might want to test this on a small image. You can create your own image by just building a 3D array of uint8. Or you can shrink an image; see help imresize. If your program is really slow, you can turn in your results on smaller versions of the assigned images, for a small loss of credit.

Hint: Sometimes as you run k-means, a cluster will wind up with no points assigned to it. Once this happens, the center of the cluster isn't defined, and the cluster will never get any new points. It is fine if at the end, k-means produces some clusters with no points. This won't affect quantization except in that you will be wasting some clusters. But you have to be careful that you handle this situation in a way that doesn't produce other errors. An even better solution would be to remove a cluster once this happens and add a new cluster to replace it. You can implement any approach, as long as your program produces reasonable results and you document your decision.

6. **Challenge Problem 15 points:** Color Segmentation. This problem is getting a little ahead of ourselves, because we haven't talked about segmentation yet, but hey, it's a challenge problem.

   The class web page shows six images of leaves on plain backgrounds. Your task is to use K-means to separate each leaf from its background. You should write a function M = segment_leaf(I). I will be one of the leaf images. M will be a binary matrix (all 0s and 1s). If M(i,j) == 1, this means that pixel (i,j) is a leaf pixel, otherwise it is a background pixel.

   One way to approach this problem is by applying K-means to the images, to divide the pixels into two groups by their color. One group may correspond to the leaf, and one to the background. You will also need to figure out which group is the leaf, and which is the background. If you use some parameters, you should use the same parameters on all six leaves. In your writeup, show the result of your program on all six images. A relatively straightforward implementation of this idea will work pretty well on some of the images, and not so well on others. You will get some partial credit for doing this. For full credit, try to build on this to handle problems like highlights, shadows, and small leaves. There is a link to a paper on the web site that will give you some ideas of what might work, as well as explaining why you might want to separate leaves from their background.

   Hint: If your programs are slow, it should be ok to shrink images (see imresize) before trying to segment them.