ECE 592 Project 1

Kudiyar Orazymbetov (korazym@ncsu.edu)
Nico Casale (ncasale@ncsu.edu)

September 25, 2017

Note that the entries are links.

Contents

List of Tables

List of Figures

Listings

1 Introduction

This project covers an implementation and study of image compression using *k-means* clustering. We show that an image can be reconstructed from representative patches that are learned from its particular distribution of pixels. Qualitatively, the reconstruction is not particularly unappealing and the quantitative results are not poor, either.

As a further experiment, we generated our own implementation of *k-means* in MATLAB that is qualitatively similar to the output of MATLAB's built in `kmeans(.)` function but does not yet compete in terms of time to execute and quantitative distortion values.

The code used to execute this project can be found in Appendix A.

2 Loading an image

For the project we chose an image of size 512x512, pictured below. Note that we used a grayscale image for a low-complexity implementation that computes quickly. If we used a color image, the *k-means* algorithm would be passed a matrix that was $3\times$ as large, causing it to converge slowly.

Figure 2.1: One of the images we used for the project.

3 Reconstruction of an Image with Assigned Clusters

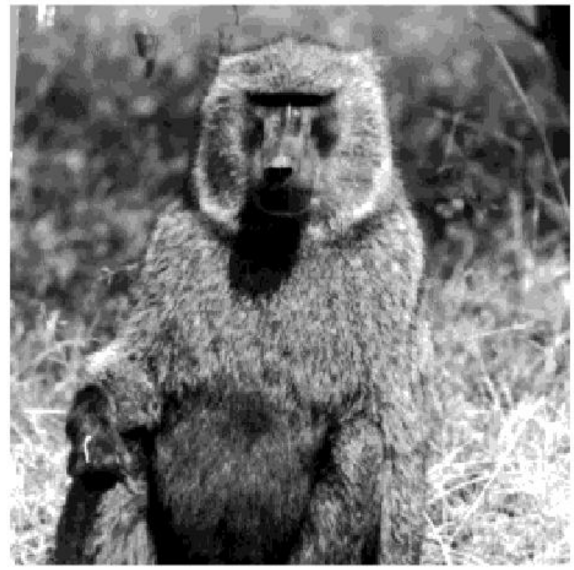

Figure 3.1: The original image (left) with a quantized version (right).

Qualitatively, the image is recognizable and not dramatically distorted by the reconstruction. This image is the result of $P = 2$ and $K = 16$.

4 Rate vs. Distortion Performance

For this part, we chose rate R to be $[0.25, 0.5, 0.75, 1]$. To find the number of clusters given a certain rate, we used the following:

$$C = round(2^{RP^2}) \tag{4.1}$$

Where C is the number of clusters, R is the coding rate, and P is the patch size (in one dimension only.)

The corresponding number of clusters to the rates given is $[2, 4, 8, 16]$ for $P = 2$. The figure below illustrates the results of these variations.

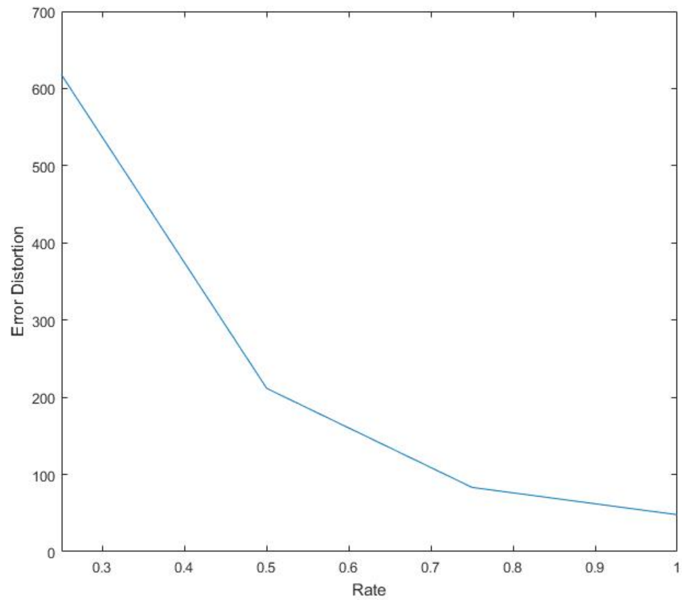

Figure 4.1: Rate (R) and Distortion (D) performance for patch size $P = 2$

The plot shows that as we increase the rate, distortion decreases. This corresponds to an increase in cluster numbers, which allows the *k-means* algorithm to more accurately represent the subtleties of the image.

5 Varied Patch Sizes

As our image size is 512x512, the maximum number of clusters, which is power of 2, for the case of $P = 4$ is 16384. If we used 16384 clusters, each patch of the image could be represented by its unique patch, which corresponds to the rate of 0.75. Because we cannot represent the image in more clusters than actually exist in the image, we only attempted rates from $[0.25, 0.5, 0.75]$. Below is a graph of the original RD plot with the new RD values for $P = 4$.

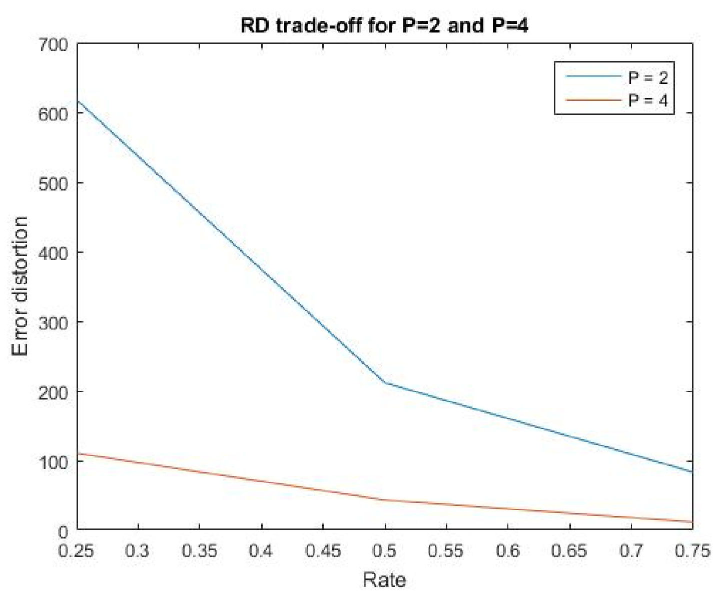

Figure 5.1: Rate (R) and Distortion (D) performance for patch size $P = 2,\ 4$

From the plot, we can see that even with a small rate increase we can substantially decrease the error distortion. It has a steeper drop in error when we increase the rate from 0.25 to 0.5 compared to the increase from 0.5 to 0.75. The underlying reason of getting less error with a patch size of 4 is we are having 16 components for each patch. Thus, the *k-means* algorithm has more dimensions to calculate the distance to other patches. Each cluster can convey a wider range of values and capture fine-grained detail in pixel change across patch boundaries. When $P = 2$ there are only 4 pixels in each patch, so the expressive capabilities of *k-means* clustering is limited.

6 Better Compression with Entropy Coding

Being that some clusters are used more frequently to represent patches in the image, we can reduce the coding length for more likely clusters. This is similar to Morse code, where the most common letters are represented in the smallest numbers of dots and dashes. For different coding rates R, which correspond to the number of clusters as Eq. (4.1), we noticed that we could reduce the number of bits required to represent each cluster (in a normalized measure) in each case.

R (Coding Rate)	**Normalized Coding Length (bits per pixel)**
0.25	0.250
0.5	0.482
0.75	0.735
1	0.962

Table 1: Coding Rate vs. Normalized Coding Length with Entropy Coding.

From above we can conclude the bits we use per pixel is smallest when we have a smaller cluster size ($C = 2$). As we increase the cluster size, we need to use more bits to represent each cluster. Nevertheless, we can decrease the bits for 1 to 0.96 when cluster size is 16. In this configuration, we can achieve a 4% reduction in coding rate over an implementation that does not use *entropy coding*.

7 Additional Direction: MATLAB implementation of K-Means

To develop an implementation of *k-means* in MATLAB, we referenced Wikipedia's article on K-means.

The *k-means* algorithm (*a.k.a.* Lloyd's algorithm) has three main steps:

1. Initialize the k centroids, or cluster locations.
2. Assign each sample of the data to one of the centroids. The centroid with the smallest squared Euclidian distance to the sample is chosen.
3. Update each cluster as the centroid (geometric average in P^2-Dimensional space, where P is the patch dimension) of the samples that were assigned to it in step 2.
4. repeat steps 2 and 3 until the assignments no longer change.

Note that *k-means* is not guaranteed to find the global optimum clustering, as it is highly dependent on step 1. The initialization is the primary factor in the convergence and quality of the clustering operation. Some notes on initialization are in the next section.

Below is an example of the output of our *k-means* function on a $[400 \times 400]$ image with a patch size of $[2 \times 2]$ and 16 clusters.

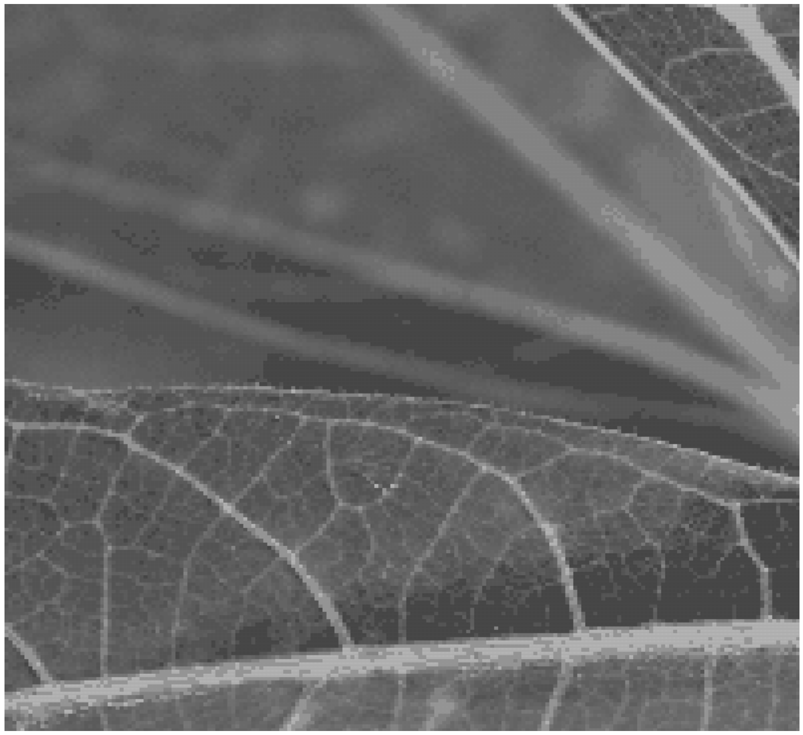

Figure 7.1: Example output of `kmeans_alt(.)`.

7.1 Initialization of *k-means*

At first, we took the following approach to initialize the clusters:

1. Find the global minimum and maximum of the pixels in the image.
2. Generate K linearly spaced points in $\mathbb{R}^{1\times P}$.

However, this approach took ∼50 iterations to converge. Likewise, because it didn't take into account the statistical distribution of the pixels in the image, it couldn't capture the subtleties of the image where most of its pixels lay.

After looking at the Wikipedia page for *k-means*, we noticed that there was some discussion on initialization methods. As a second approach, we tried the Forgy Method, which takes k random samples from the image and uses them as the initial clusters. This method was much more effective as it reduced the number of iterations needed to converge to ∼15. As this method is inherently stochastic, some assays would take longer to converge as they were initialized to locations that were distant from the more stable regions. We describe good locations for clusters as 'stable regions' because there are an infinite number of clusters and locations that could converge under Lloyd's algorithm.

7.2 Timing and Distortion Comparisons

The table below illustrates some comparisons between MATLAB's built in *k-means* function and the one we coded. All values were averaged over 20 iterations in MATLAB R2016b. The computer used was a Dell laptop with an Intel Core i3 processor.

	MATLAB	Our Implementation
Average time	0.863	2.578
Minimum time	0.421	0.965
Distortion	27.386	56.191

Table 2: Comparison of MATLAB's *k-means* and ours.

Note that our function is both slower and less accurate than MATLAB's by a factor of ~2. In the future, improvements could be made to the initialization procedure to try to get a more accurate representation that takes less time to compute. From a purely computational perspective, the *k-means* exhibits a great deal of parallelism. The result of step 3 is strictly dependent on step 2, but each is parallelizable on its own. In the next section, some details on a potential parallelization of *k-means* are discussed.

7.3 Profiler Results

In analyzing the bottlenecks of the code we wrote to implement *k-means*, we noticed that there is one computationally expensive part of the calculation that is hard to improve outside of parallelization. The figures below illustrate the most time-consuming lines of our function. Note that this screenshot is from a run of our alternate *k-means* function that converged in 9 iterations, a relative minimum given our current implementation.

Profile Summary

Generated 24-Sep-2017 16:31:51 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
kmeans_alt	1	1.993 s	0.006 s	
assignmentStep	9	1.985 s	1.985 s	
updateStep	9	0.002 s	0.002 s	

Self time is the time spent in a function excluding the time spent in its child functions.

Figure 7.2: First page of MATLAB's profiler results.

assignmentStep (Calls: 9, Time: 1.985 s)

Generated 24-Sep-2017 16:32:58 using performance time.

function in file C:\Users\ncasa\Documents\GitHub\ece592_p1\project1\code\assignmentStep.m

Parents (calling functions)

Function Name	Function Type	Calls
kmeans_alt	function	9

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
34	`[~,newId] = min(a2); % most ex...`	360000	1.082 s	54.5%	
33	`a2 = sum(a1,2); % mid-expensiv...`	360000	0.674 s	34.0%	
32	`a1 = (patches(i) - Cn).^2; % 1...`	360000	0.085 s	4.3%	
37	`Cnew(newId,:) = Cnew(newId,:) ...`	360000	0.073 s	3.7%	
38	`count(newId) = count(newId) + ...`	360000	0.024 s	1.2%	
All other lines			0.047 s	2.4%	
Totals			1.985 s	100%	

Figure 7.3: Analysis of the `assignmentStep(.)` function.

As shown in the figure, our two most expensive lines deal with the assignment of samples to clusters. In order to determine which of the current clusters a sample belongs to, the code takes the squared Euclidian distance of each point to each of the clusters. The cluster that is closest to the point is assigned. This step is computationally expensive because the function `assignmentStep(.)` must sum along the y dimension the distances between each of the pixel values in the patch and the centroids. Then, the `min(.)` function has to find the smallest point in the

$[K \times 1]$ array. These calls are necessary for quick convergence but negatively impact the time it takes to reach convergence. We tried to use ersatz methods, such as taking the absolute value of the distances instead of squaring them, or taking only the 1^{st} column of distances, but these approaches led to less accurate assignment. Even though they were less computationally expensive individually, they led to a higher number of iterations, which totally counteracted the positive effects of the individual computation.

If we were to implement *k-means* on a GPU, we could parallelize the thousands of calls to `assignmentStep(.)` so that the function would run faster. That, paired with a more effective initialization, would create a good implementation of *k-means* that could be on par with MATLAB.

8 Conclusion

In conclusion, we have found that it is possible to compress an image using clustering algorithms such as *k-means*. The quality of the reconstruction is dependent on the patch size P and the number of clusters C (`K` in our MATLAB code.) There is also a reasonable level of additional compression that could be added if we used entropy coding to represent more common clusters with less bits. Finally, we have seen that a capable implementation of *k-means* is possible and not especially complex.

9 Appendix A: Code Listings

9.1 Code for Parts 2-5 of the Project

9.1.1 `mainCode.m`

Listing 1: Code to solve parts 2-5 of the Project.

```
%{
ECE 592 Project 1

Kudiyar (Cody) Orazymbetov
korazym@ncsu.edu

Nico Casale
ncasale@ncsu.edu
%}

%% PART 1
clear; close all;
addpath('utility', 'kmeans');
setup();
global imagesFolder
overwriteImage = 0;

fprintf('ECE 592 Project 1\n');
fprintf(strcat(datestr(now),'\n'));
% read image
I = double(imread('image3.gif'));
[M, N] = size(I);

%% PART 2
P = 2; % square patch dimension
R = 0.25:0.25:1; % various values for Rate
K = round(2 .^(R*P^2));
for j = 1:length(K)
   % partition into patches
   Ipartitioned = im2col(I, [P P], 'distinct');

   % apply k-means
   [idx, Cn] = kmeans(Ipartitioned', K(j));

   % reconstruct image
   indexrepresentations = zeros(length(idx), P^2);
   for i = 1:length(idx)
      indexrepresentations(i, :) = Cn(idx(i), :);
   end
   Iquantized = col2im(indexrepresentations', [P P], size(I), 'distinct');

   subplot(1, 2, 1);
   imshow(I, []);
   f2 = subplot(1,2,2);
   imshow(Iquantized, []);

   % Calculate distortion D
   D(j) = sum(sum((I - Iquantized) .^2)/(M*N));
end
```

```

%% PART 3 Rate & Distortion Plot
R = 0.25:0.25:1;
plot(R,D)

%% PART 4
P = 4;
R = 0.25:0.25:0.75;
K = round(2 .^(R*P^2));
for j = 1:length(K)
   % partition into patches
   Ipartitioned = im2col(I, [P P], 'distinct');

   % apply k-means
   [idx, Cn] = kmeans(Ipartitioned', K(j));

   % reconstruct image
   indexrepresentations = zeros(length(idx), P^2);
   for i = 1:length(idx)
      indexrepresentations(i, :) = Cn(idx(i), :);
   end
   Iquantized = col2im(indexrepresentations', [P P], size(I), 'distinct');

   % compute distortion D
   D1(j) = sum(sum((I - Iquantized) .^2)/(M*N));
end

% Plot RD trade-off for P =2 and P =4;
figure
R = 0.25:0.25:0.75;
plot(R(1:3),D(1:3))
title('RD trade-off for P=2 and P=4')
xlabel('Rate')
ylabel('Error distortion')
hold on
plot(R,D1)
hold off
legend('P = 2', 'P = 4')

%% PART 5
% First run Part 2, then run this part

P = 2; % square patch dimension
R = 0.25:0.25:1;
K = round(2 .^(R*P^2));

for j = 1:length(K)
   % partition into patches
   Ipartitioned = im2col(I, [P P], 'distinct');

   % apply k-means
   [idx, Cn] = kmeans(Ipartitioned', K(j));

   % recontstruct image
   indexrepresentations = zeros(length(idx), P^2);
   for i = 1:length(idx)
```

```
      indexrepresentations(i, :) = Cn(idx(i), :);
   end
   Iquantized = col2im(indexrepresentations', [P P], size(I), 'distinct');

   % plot results
   subplot(1, 2, 1);
   imshow(I, []);
   f2 = subplot(1,2,2);
   imshow(Iquantized, []);

   % compute distortion
   D(j) = sum(sum((I - Iquantized) .^2)/(M*N));

   for k = 1:K(j)
      Np(k) = sum(idx == k);
   end

   H = 0;
   for m = 1:K(j)
      H = H -(Np(m)*P^2/(M*N)*log2(Np(m)*P^2/(M*N)));
   end

   r(j) = H/P^2; % look at r array to see rates
end
```

9.2 Code for our implementation of K-means

9.2.1 `compare_Kmeans.m`

Listing 2: Code to compare our implementations of *k-means*.

```
%{
ECE 592 Project 1

Kudiyar (Cody) Orazymbetov
korazym@ncsu.edu

Nico Casale
ncasale@ncsu.edu

This file is used to compare the built-in kmeans function and
the function generated by Cody and Nico for project 1.
%}


clear; close all;
addpath('../utility', '..');
global seed
seed = 475859;
rng(seed);
global imagesFolder
imagesFolder = '../../images';
addpath(imagesFolder);
overwriteImage = 0;

fprintf('ECE 592 Project 1\n');
```

```
fprintf(strcat(datestr(now),'\n'));

% read image
sz = 400;
I = double(rgb2gray(imcrop(imread('cassava.jpg'),[50 50 sz-1 sz-1])));
[M, N] = size(I);
%image(I) properties
%whos I
f = figure(1);
I3 = cat(3, I, I, I);
image(uint8(I3));
prettyAxes();
prettyPictureFig(f); prettyPictureFig(f);

P = 2; % square patch dimension

% partition into patches
Ipartitioned = im2col(I, [P P], 'distinct');

%% MATLAB's kmeans
% apply k-nearest neighbor
K = 16;
tic;
[idx, Cn] = kmeans(Ipartitioned', K);
toc;

% recreate image with the clusters learned by kmeans
indexrepresentations = zeros(length(idx), P^2);
for i = 1:length(idx)
   indexrepresentations(i, :) = Cn(idx(i), :);
end

Iquantized = col2im(indexrepresentations', [P P], size(I), 'distinct');

f = figure(2);
quantizedI3 = cat(3, Iquantized, Iquantized, Iquantized);
image(uint8(quantizedI3));
prettyAxes();
prettyPictureFig(f);prettyPictureFig(f);


%% Nico and Cody's kmeans
profile on
tic
[idx_alt, Cn_alt] = kmeans_alt(Ipartitioned', K);
toc
profile viewer

indexrepresentations_alt = zeros(length(idx_alt), P^2);
for i = 1:length(idx_alt)
   indexrepresentations_alt(i, :) = Cn_alt(idx_alt(i), :);
end

Iquantized_alt = col2im(indexrepresentations_alt', [P P], size(I), 'distinct');

f = figure(3);
```

```
quantizedI3_alt = cat(3, Iquantized_alt, Iquantized_alt, Iquantized_alt);
image(uint8(quantizedI3_alt));
prettyAxes();
prettyPictureFig(f);prettyPictureFig(f);

compare = [Cn Cn_alt];

D_kmeans = sum(sum((I(:,:,1) - quantizedI3(:,:,1)) .^2)/(M*N))
D_kmeans_alt = sum(sum((I(:,:,1) - quantizedI3_alt(:,:,1)) .^2)/(M*N))

%% Get an average time and distortion for each
REPS = 20;
tMin = inf;
tic;
for rep = 1:REPS
   fprintf('rep = %d\n', rep);
   rng shuffle
   tSt = tic;
   [idx, Cn] = kmeans(Ipartitioned', K);
   tElapsed = toc(tSt);
   tMin = min(tElapsed, tMin);

   D_kmeans(rep) = sum(sum((I(:,:,1) - quantizedI3(:,:,1)) .^2)/(M*N));

end
tAvg = toc/REPS;
fprintf('MATLAB: min = %d, avg = %d\n', tMin, tAvg);

REPS = 20;
tMin_alt = inf;
tic;
for rep = 1:REPS
   fprintf('rep = %d\n', rep);
   rng shuffle
   tSt = tic;
   [idx_alt, Cn_alt] = kmeans_alt(Ipartitioned', K);
   tElapsed_alt = toc(tSt);
   tMin_alt = min(tElapsed_alt, tMin_alt);

   D_kmeans_alt(rep) = sum(sum((I(:,:,1) - quantizedI3_alt(:,:,1)) .^2)/(M*N));

end
tAvg_alt = toc/REPS;
fprintf('592: min = %d, avg = %d\n', tMin_alt, tAvg_alt);

D_kmeans_avg = mean(D_kmeans);
D_kmeans_avg_alt = mean(D_kmeans_alt);
fprintf('Distortion - MATLAB = %d, 592 = %d\n', D_kmeans_avg, D_kmeans_avg_alt);

%% save images
file = sprintf('altkmeans_reconstructed');
file = strcat(imagesFolder, file);
print(file, '-dpng');
```

9.2.2 `kmeans_alt.m`

Listing 3: Our implementation of *k-means*.

```
%{
ECE 592 Project 1

Kudiyar (Cody) Orazymbetov
korazym@ncsu.edu

Nico Casale
ncasale@ncsu.edu

This is our own implementation of K-means
https://en.wikipedia.org/wiki/K-means_clustering#Standard_algorithm
%}

function [idx, Cn] = kmeans_alt(patches, K)

   %{
   Generate initial centroids by choosing
   K random samples of patches
   %}
   inds = randi(length(patches), [K, 1]);
   Cn = patches(inds,:);

   % Group and update centroids
   iter = 0;
   max_iter = 100; % maximum iterations

   change = inf;
   idx = zeros(length(patches),1);

   while (change && iter < max_iter)
      [Cnew, count, idx] = assignmentStep(Cn, idx, patches);
      Cnew = updateStep(Cnew, count);
      change = max(max(abs(Cnew - Cn)));
      Cn = Cnew;
      iter = iter + 1;
   end

end
```

9.2.3 `assignmentStep.m`

Listing 4: Helper function for `kmeans_alt(.)`.

```
%{
ECE 592 Project 1

Kudiyar (Cody) Orazymbetov
korazym@ncsu.edu

Nico Casale
ncasale@ncsu.edu

Helper function for kmeans_alt

Assigns each patch to a cluster,
```

```
Sums continuously the new centroid, to be averaged later
Counts the number of patches in each cluster
%}

function [Cnew, count, idx] = assignmentStep(Cn, idx, patches)

   %{
   assign each patch to one of the K centroids in Cn
   based on minimum distance to a given centroid
   %}

   % todo: repmat on Cn to make more parallel?

   % initialize new clusters
   Cnew = zeros(size(Cn));
   count = zeros(size(Cn,1),1);

   for i = 1:length(patches)
      % broken up step by step to find bottleneck
      a1 = (patches(i) - Cn).^2; % least expensive
      a2 = sum(a1,2); % mid-expensive
      [~,newId] = min(a2); % most expensive

      idx(i) = newId;
      Cnew(newId,:) = Cnew(newId,:) + patches(i,:);
      count(newId) = count(newId) + 1;
   end

end
```

9.2.4 `updateStep.m`

Listing 5: Helper function for `kmeans_alt(.)`.

```
%{
ECE 592 Project 1

Kudiyar (Cody) Orazymbetov
korazym@ncsu.edu

Nico Casale
ncasale@ncsu.edu

Helper function for kmeans_alt

Updates each cluster as the centroid of the patches
assigned to the cluster at hand
%}

function Cnew = updateStep(Cnew, count)

   % average
   for i = 1:length(Cnew)
      % don't modify the cluster if no one belongs to it
      if count(i) == 0
         continue;
```

```
        end
        Cnew(i,:) = Cnew(i,:)/count(i);
    end
end
```