CS 231A Computer Vision, Spring 2016
Problem Set 4

Due Date: May 20th 2016 11:59 pm

1 Viewpoint Estimation With Bag of Words

In the previous problem set, we explored object detection with histogram of oriented gradients and support vector machines. Often, we not only care about detecting the object, but also recognizing its pose. In this problem, we will build a classifier for estimating one of 16 orientations of different cars, using a BOW method introduced with the EPFL car dataset [1].

Download the starter code from `http://cs231a.stanford.edu/hw/ps4/viewpoint.zip`

Again, download vlfeat: `http://www.vlfeat.org/download/vlfeat-0.9.20-bin.tar.gz`. Put this in a `vlfeat` directory inside the project root folder.

If you ever want more detail or get stuck, referring to lecture notes and the CA section will help.

(a) **[10 points]** The first step is to build a dictionary of codewords which we will use to construct our Bag of Words models. To do so, run kmeans to find the centers of clusters of dense SIFT features in `generateClusterCenters.m`. Submit your code.

(b) **[20 points]** The next step is to now build our bag of words per image. To do so, we simply compute a histogram of codewords occurencies in the image as discussed in lecture. We then repeat this using a spatial pyramid of different grid sizes. Implement this in `generateBOWFeatures.m`. As a sanity check, using a subset of the training data, you should, on most runs, get above 20% classification accuracy. Submit your code.

(c) **[5 points]** Now train and run the SVM classifier using all the training examples in `viewpointEstimation.m`. Because we're working with a lot of images, be prepared for this to take some time. You should see an accuracy above 65% on the test set, well above random chance ($\frac{1}{16}$). Submit your train and test accuracies.

(d) **[5 points]** Let's explore the properties of this Bag of Words model. What happens if we don't use a spatial pyramid (set `pyramid_depth=1`) to augment our feature set? What happens as we decrease the number of words (set `num_centers=2`) in our Bag of Words? Explain your results.

[1] `http://cvlabwww.epfl.ch/~lepetit/papers/ozuysal_cvpr09.pdf`

Solution:

(a) generateClusterCenters.m

```
function [ centers ] = generateClusterCenters( train_data, num_centers,
    num_subset_images )
%GENERATECLUSTERCENTERS Summary of this function goes here
% Inputs
%   train_data - an N x 3 matrix, with each row being a tuple of
%                (image_id, viewpoint degree, degree bucket)
%   num_centers - the number of centers (words in the BOW)
%   num_subset_images - the number of subset images
%
% Output
%   centers - the centers (words in the BOW)
%
% Algorithm
% The basic idea is to take all the feature vectors in the training data
% and then cluster them. The centers of the cluster will form the "words"
% in our Bag of Words representation. Knowing this, we do:
% (1) The amount of feature vectors in all the training images is too big.
%     Instead, lets choose a random subset of num_subset_images of the
%     training images to cluster on.
% (2) For each image in this subset, compute its dense SIFT features.
%     vl_dsift does this.
% (3) Run K-means (vl_kmeans) on the aggregated feature vectors to come up
%     with the num_centers cluster centers

% Useful code to get the image filepath with image_id:
% pathname = sprintf('car_epfl/images/%04d.jpg', <IMAGE_ID>);

all_d = [];
vec = randsample(size(train_data,1),num_subset_images);
for j = 1:numel(vec)
    i = vec(j);
    pathname = sprintf('car_epfl/images/%04d.jpg',train_data(i,1));
    im = imread(pathname);
    im = im2single(rgb2gray(im));
    [f,d] = vl_dsift(im);
    all_d = [all_d d];
end
[centers, assignments] = vl_kmeans(single(all_d), num_centers);
end
```

(b) generateBOWFeatures.m

```
function [ features ] = generateBOWFeatures( im, centers, pyramid_depth)
%GENERATEBOWFEATURES Generates a feature vector for an image using dense
% SIFT Bag of Words
%
% Inputs
%   im - the image
%   centers - the centers from KMeans
%   pyramid_depth - the max spatial pyramid depth
%
% Ouput
%   features - the BOW features (nbins*hist_count by 1 vector)
%
% Algorithm
% (1) Compute the dense SIFT features for the image. Note that dense SIFT
%     from vlfeat reduces the dimensions slightly (by 9) because of the
%     sliding window used in its computation (similar to HOG). Also note
%     that each pixel (except the omitted ones near the border) will have a
%     dense SIFT feature associated with it.
% (2) For each pyramid depth d in 1:pyramid_depth, divide the original image
%     into a  d-by-d grid of sub-images.
```

```
% (3) For each subimage, then take each dense SIFT feature vector within the
%     subimage and find the nearest center closest to that vector.
% (4) Build a histogram of the distribution of closest centers (this is
%     your bag of words).
% (5) Concatenate all subimage histograms together in a specific order. Our
%     order is going left-to-right, then top-down across the subimages. We
%     store depth 1, then 2, then etc.
nbins = size(centers,2);
hist_count = 0;
for i=1:pyramid_depth
    hist_count = hist_count + i*i;
end
features = zeros(nbins*hist_count,1);
h = size(im,1); w = size(im,2);
bin_idx = 0;
[f,d] = vl_dsift(im);

d = double(d);
[~,assignments] = max(bsxfun(@minus,centers'*d,dot(centers,centers,1)'/2),[],1);
for b=1:pyramid_depth
    division = b;
    for j=1:division
        for k=1:division
            hist = zeros(1,nbins);
            lower_h = (j-1)*h/division +1;
            upper_h = j*h/division;
            lower_w = (k-1)*w/division +1;
            upper_w = k*w/division;
            h_idx = find(f(2,:) >= lower_h & f(2,:) <= upper_h);
            w_idx = find(f(1,:) >= lower_w & f(1,:) <= upper_w);
            idx = intersect(h_idx,w_idx);
            tmp_assign = assignments(idx);
            for q = 1:nbins
                hist(q) = sum(tmp_assign==q)/size(tmp_assign,2);
            end
            features((bin_idx*nbins+1):(nbins*(bin_idx+1))) = hist;
            bin_idx = bin_idx + 1;
        end
    end
end

end
```

(c) We obtained a training accuracy of 0.9907 and a testing accuracy of 0.6786.

(d) If we don't use a spatial pyramid, we lose the ability to capture the spatial distribution of codewords and reduce the discrimination power of our classifier resulting in lower testing accuracy. We will only be able to consider the aggregate histogram across the whole image. Thus, we would not be able to distinguish between two images with different histograms in the left and right half of the image that sum to the same histogram across the image. If we decrease the number of words, then the testing accuracy will also decrease because the words will not be as representative of the appearance distribution within and across classes. Distinguishing codewords that should be clustered separately will be merged if an insufficient number of words is specified.

2 Image Segmentation

Image segmentation is the process of partitioning an image into meaningful structures. Two famous segmentation methods are meanshift [2] and normalized cut [3]. In this problem, you will implement significant parts of both methods.

Download the starter code from http://cs231a.stanford.edu/hw/ps4/segmentation.zip

If you ever want more detail or get stuck, referring to lecture notes and the CA section will help.

Meanshift

(a) **[25 points]** Complete image segmentation using meanshift in `meanshift.m`. Submit your code and the resulting segmentation. View your segmentation results using `segmentation.m`.

(b) **[5 points]** Try the different images provided (`plates.jpg` and `rocks.jpg`). Play around with the bandwidth parameter to get good results on these images and explain the effect of changing it. Also provide your segmentations on these images (try to get the best you can, but don't spend too much time).

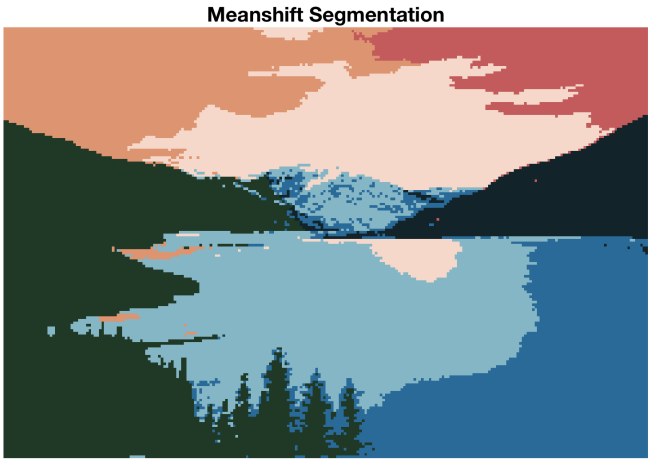
[2] https://courses.csail.mit.edu/6.869/handouts/PAMIMeanshift.pdf
[3] http://www.cs.berkeley.edu/~malik/papers/SM-ncut.pdf

Solution:

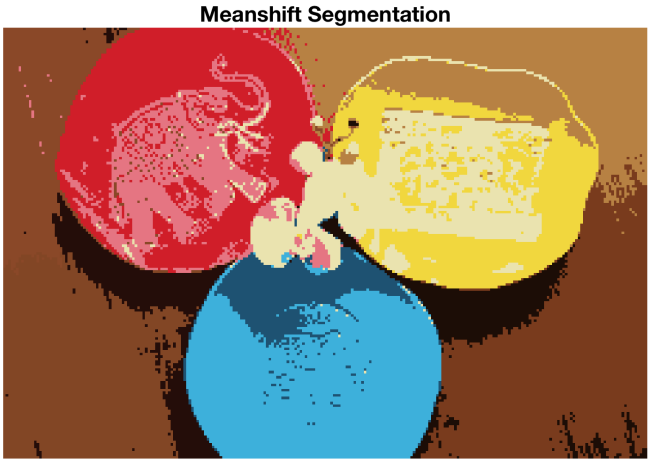
(a) meanshift.m

```
function pixel_clusters  = meanshift(im, features, bandwidth)
%MEANSHIFT: Image segmentation using meanshift
% Arguments:
%    im - the image being segmented (H by W by 3 matrix)
%    features - M by #pixels matrix that are the M features associated with
%       each pixel
%    bandwidth - A parameter that determines the radius of what particpates
%       in the mean computation
% Returns:
%    pixel_clusters - H by W matrix where each index tells what cluster the
%       pixel belongs to. The clusters must range from 1 to N, where N is
%       the total number of clusters.
%
% The meanshift algorithm can be done in the following steps:
% (1) Keep track of an array whether we have seen each pixel or not.
% Initialize it such that we haven't seen any.
% (2) While there are still pixels we haven't seen do the following:
%       - Pick a random pixel we haven't seen
%       - Until convergence (mean is within 1% of the bandwidth of the old
%       mean), mean shift. For each iteration of the meanshift, if a pixel
%       is within the bandwidth circle, then it should be marked as seen
%       - If it's sufficienctly close (within half a bandwidth) of another
%       cluster, say it's part of that cluster
%       - If it's not sufficiently close to any other cluster, make a new
%       cluster
% (3) After finding all clusters, assign every pixel to the nearest cluster
% in feature space.

num_pts = size(features, 2);
bandSq          = bandwidth^2;
threshold      = 1e-2*bandwidth;                          %when mean has converged

hasVisited = zeros(1,num_pts);
ptsRemaining = find(hasVisited == 0);
cluster_centers = [];

while length(ptsRemaining) ~= 0
    tempInd         = ceil( length(ptsRemaining)*rand);        %pick a random seed
        point
    stInd           = ptsRemaining(tempInd);                   %use this point as
        start of mean
    currentMean          = features(:,stInd);                           % intilize
        mean to this points location

    while 1
        oldMean = currentMean;
        sqDistToAll = sum((repmat(currentMean,1,num_pts) - features).^2);
        inInds      = find(sqDistToAll < bandSq);               %points within
            bandwidth
        currentMean      = mean(features(:,inInds),2);
        hasVisited(inInds) = 1;


        if norm(currentMean-oldMean) < threshold

            %check for merge posibilities
            mergeWith = 0;
            for cN = 1:size(cluster_centers,2)
                distToOther = norm(currentMean-cluster_centers(:,cN));     %
                    distance from posible new clust max to old clust max
                if distToOther < bandwidth/2                    %if its within
                    bandwidth/2 merge new and old
                    mergeWith = cN;
```

```
                    break;
                end
            end

            if mergeWith == 0
                cluster_centers = [cluster_centers currentMean];

            end
            break;
        end
        ptsRemaining = find(hasVisited == 0);
    end
end

pixel_clusters = zeros(1,size(features,2));

for p = 1:size(features,2)
    closestInd = 1;
    smallestDist = norm(features(:,p) - cluster_centers(:,1));
    for c = 2:size(cluster_centers,2)
        d = norm(features(:,p) - cluster_centers(:,c));
        if d < smallestDist
            smallestDist = d;
            closestInd = c;
        end
    end
    pixel_clusters(p) = closestInd;

end
pixel_clusters = reshape(pixel_clusters, [size(im,1) size(im,2)]);
```

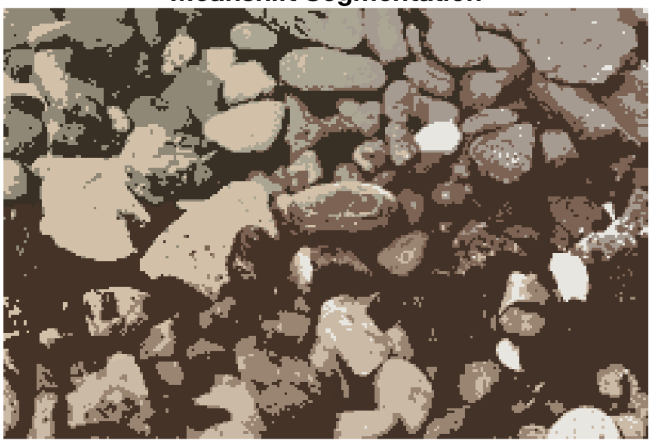


(b)



Making the bandwidth parameter bigger means using a larger radius in feature space when thresholding to decide which pixels to use. So each iteration of calculating the mean will have more pixels included in the calculation. If the bandwidth is increased then the different modes in our feature distribution are increasingly likely to be identified as a single mode, i.e., fewer cluster centers and fewer segments.

Meanshift Segmentation



Normalized cut

Below, we will both discuss the inspiration for and outline the major steps in segmenting an image by normalized cuts. Feel free to read the paper for the complete mathematical derivation.

The idea for normalized cuts stems from graph theory, where we treat the image as a connected graph $G = (V, E, W)$, with V being the node set, E being the edge set, and W being the matrix of weights for each edge. In this case, the pixels are the nodes and weighted edges are some function of the pixel features. The general min-cut problem assumes we have some node set $V = \{1, ..., n\}$ that we want to partition into subsets A and B for clustering purposes. For segmentation, a minimum cut generally does not work, because it will usually divide A and B into subsets with 1 node and $n - 1$ nodes respectively, which is usually a suboptimal clustering scheme. On the other hand, normalized cuts compute the cut cost as a fraction of each subgroup's total edge connections to the rest of the nodes in the graph. Therefore, normalized cuts favor a more evenly distributed partitioning when minimized.

We are given W, a weight matrix where w_{ij} is the weight of the edge between nodes i and j, and we compute D, the diagonal matrix where $D_{i,i} = \sum_j w_{ij}$. Let x be a vector where $x_i = 1$ if $i \in A$ and $x_i = -1$ if $i \in B$. Then we define

$$k = \frac{\sum_{x_i>0} D_{i,i}}{\sum_i D_{i,i}}.$$

Intuitively, k is the sum of weights of all the edges in A over the total weight of the graph. Next we define

$$b = \frac{k}{1-k},$$

which is the proportion of the weights of A versus the weights of B. We use this to define

$$y = (1 + x) - b(1 - x),$$

which allows us to set up the facts that

$$y^T D1 = 0 \text{ and } y^T Dy = b1^T D1.$$

Using the previous defintions, we can derive that

$$\min_x \text{ncut}(x, W, D) = \min_y \frac{y^T(W - D)y}{y^T Dy}$$

subject to the constraints

$$y_i \in \{1, -b\},\ y^T D1 = 0.$$

This formulation gives us a simpler way to find the partition that minimizes the normalized cut. By solving for y, which ultimately is just a reformulation of our partition indicator vector x, we are able retrieve the optimal partition A, B. We approximately solve for y by solving the eigensystem

$$(D - W)y = \lambda Dy,$$

which automatically satisfies the constraint $y^T D1 = 0$. The paper finds that the eigenvector v associated with the second smallest (in absolute value) eigenvalue of the generalized eigensystem

is the approximate solution to the minimum normalized cut problem. However, we find that it does not necessarily satisfy the other constraint $v_i = \{1, -b\}$. In the ideal case, this eigenvector takes on the two values, which makes partitioning the nodes easy. However, this eigenvector will generally take on continuous values and we need to find a splitting point ϵ such that all indices i such that $v_i > \epsilon$ are in A and otherwise in B. One can take conjectures at the value of the splitting point, such as 0 or the median, but searching for the optimal point works by far the best (use `fminsearch`). Thus, after finding the splitting point, we can compute x, k, b, and y to find the value of minimum normalized cut and its resulting partition.

(a) **[10 points]** Fill in the normalized cut computation in `ncut.m`.
Recall that ncut $= \frac{y^T(D-W)y}{y^TDy}$. We can compute y from the second smallest eigenvector v. By defining a partition value ϵ, we find that node $i \in A$ if $v_i > \epsilon$, otherwise $i \in B$. Submit your code.

(b) **[10 points]** Usually, a multi-segment segmentation is of interest, rather than a simple, bimodal segmentation (multiple objects or regions of interest in the image). Thus, we recursively repeat it on each subgroup A,B, after each cut until some criteria are satisfied. The overall normalized cut for segmentation is as follows:

1. Given a set of features, set up a weighted graph $G = (V, E, W)$, compute the weight on each edge, and summarize the information into W and D (we provide this code).
2. Solve $(D - W)x = \lambda Dx$ for eigenvectors with the smallest eigenvalues. You will find the matlab function call `eigs(D-W,D,2,'sm')`, which returns the two smallest (in absolute value) eigenvectors that solve this eigensystem, to be extremely useful.
3. Use the eigenvector with the second smallest eigenvalue to bipartition the graph by finding the partition point ϵ such that ncut is minimized. You might find the Matlab function `fminsearch` useful (initialize the partition point to be 0).
4. Decide if the current partition should be subdivided by ensuring that the ncut is below the threshold value (otherwise we will be partitioning regions that should belong to the same segment). Also check if the each subgraph is large - whether its size is greater than some lower bound pixel count.
5. Recursively repartition the segmented parts if necessary.

Implement this in `recursivePartition.m`. Submit your code.

(c) **[5 points]** View your segmentation results using `segmentation.m`. Submit the resulting segmentation.

(d) **[5 points]** Try the different images provided (`plates.jpg` and `rocks.jpg`). Play around with the ncut and pixel thresholds to get good segmentations of each and explain the effects of changing each. Also provide your segmentations on these images (try to get the best you can, but don't spend too much time).

Solution:

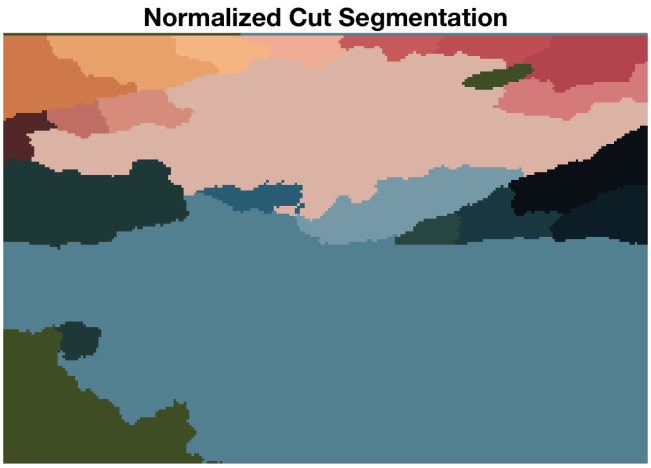
(a) ncut.m

```
function nc = ncut(partition_pt, ev2, W, D)
% NCUT: Computes the cost of the normalized cut around a particular
% partition value.
% Arguments:
%    partition_pt - The value from which we split the second smallest
%       eigenvector around
%    ev2 - The second smallest eigenvector (Nx1 vector, where N = |A|+|B|)
%    W - the similarity weight matrix
%    D - the diagonal matrix
% Returns:
%    nc - The cost of the normalized cut around the partition point
x = (ev2 > partition_pt);
x = (2 * x) - 1;
d = diag(D);
k = sum(d(x > 0)) / sum(d);
b = k / (1 - k);
y = (1 + x) - b * (1 - x);
nc = (y' * (D - W) * y) / ( y' * D * y );
end
```

(b) recursivePartition.m

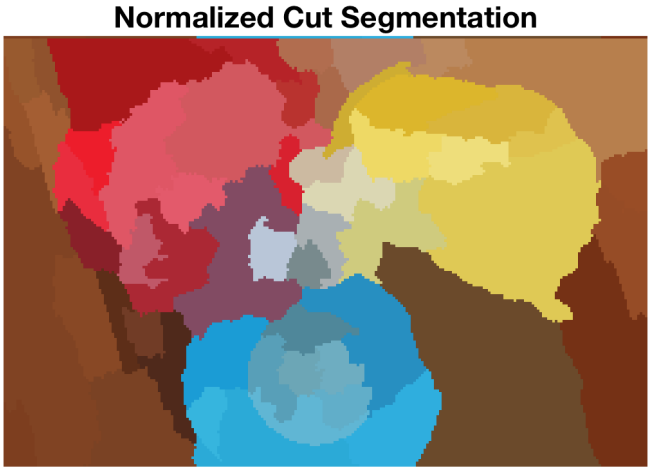
```
function seg = recursivePartition( V, W, ncut_threshold, min_pixels )
% RECURSIVEPARTITION: A recursive function that splits the graph in two
% based on the normalized cut
% Arguments:
%    V - A vector of vertex (pixel) indices.
%    W - The similarity weight matrix
%    ncut_threshold - a stopping criterion; when the normalized cut exceeds
%       a certain threshold, we stop recursively partioning.
%    min_pixels - a stopping criterion; all groups must have this minimum
%       number of pixels.
% Returns:
%    seg - The segmentation of the graph. It will be a 1D array of cells
%       where each cell represents a group. A cell contains the indices
%       (from V) of the group
%
% The approach is to compute the eigenvectors of the matrix W-D as shown in
% the paper/class materials. Then we use the mean value of this eigenvector
% as an initial guess into the best partition for the normalized cut. Then
% repeat the process on each of the new subgraphs
%
% HINT: Use the MATLAB function fminsearch() with the ncut() function to
% find the best possible partition at each step.

N = length(W);
d = sum(W, 2);
D = spdiags(d, 0, N, N);

% Step 2 and 3. Solve generalized eigensystem (D -W)*S = S*D*U (12).
[U,S] = eigs(D-W, D, 2, 'sm');

% Get the 2nd smallest eigenvector
U2 = U(:, 2);

% Partition the graph where ncut is minimized
t = 0; %mean(U2);
t = fminsearch('ncut', t, [], U2, W, D);
A = find(U2 > t);
B = find(U2 <= t);

% Step 4. Decide if the current partition should be divided
```

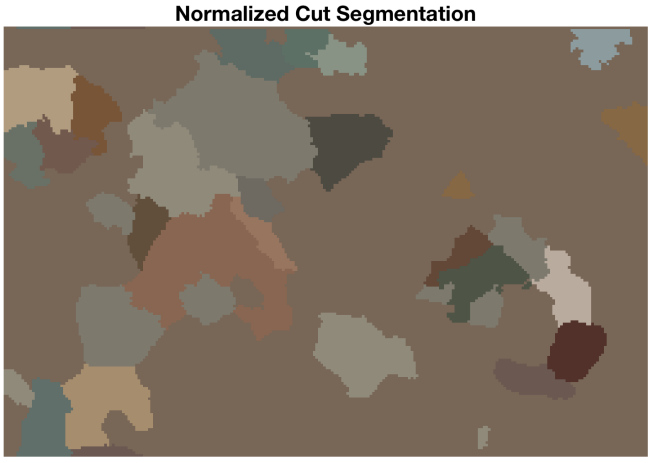
```
% if either of partition is too small, stop recursion.
% if Ncut is larger than threshold, stop recursion.
nc = ncut(t, U2, W, D);
if (length(A) < min_pixels || length(B) < min_pixels) || nc > ncut_threshold
    seg{1}   = V;
    return;
end

% segment A and B
segA = recursivePartition(V(A), W(A, A), ncut_threshold, min_pixels);
segB = recursivePartition(V(B), W(B, B), ncut_threshold, min_pixels);
seg = [segA segB];
end
```



(c)

(d) A high normalized cut cost threshold means that the algorithm will continue to cut into smaller pieces even when the cost of doing so becomes high, implying more closely related patches will be cut from one another and vice versa, a lowe normalized cut cost threshold means "closely related" patches will not be cut further as cutting a patch into two closely related segments comes at high cost. A lower min_pixels means that the algorithm will continue to cut into smaller pieces, resulting in more, smaller segments and vice versa, a larger min_pixels restricts the segments from becoming too small, so fewer recursive cuts will take place and segments will be larger.


Normalized Cut Segmentation


Normalized Cut Segmentation