Problem Set 4 Solution

CS231A: Computer Vision
Stanford University
Spring 2017

Chi Zhang
SUID: 06116342
Date: June 2, 2017

1 Face Detection with HoG

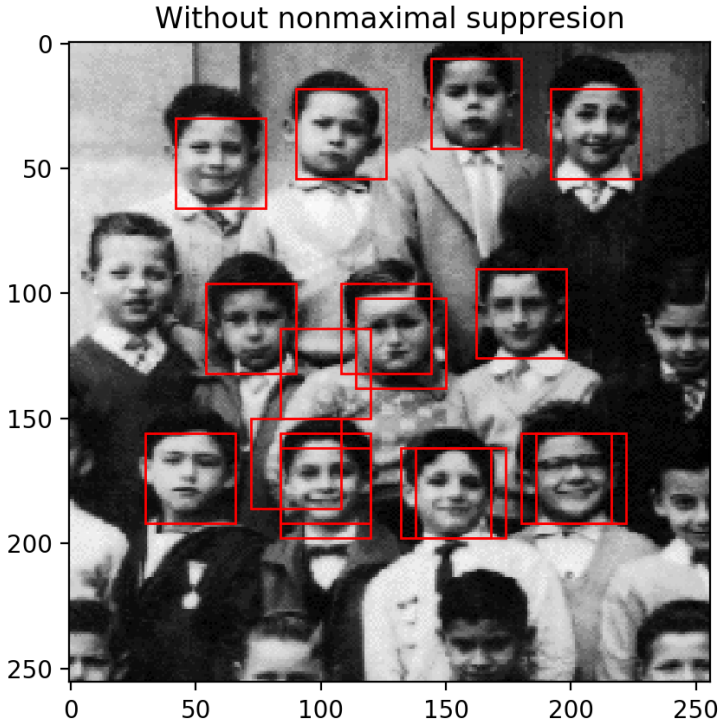
The following code includes implementations of `run_detector()`, `non_max_suppression()`:

```
'''
RUN_DETECTOR Given an image, runs the SVM detector and outputs bounding
boxes and scores

Arguments:
    im - the image matrix

    clf - the sklearn SVM object. You will probably use the
        decision_function() method to determine whether the object is
        a face or not.
        http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

    window_size - an array which contains the height and width of the sliding
        window

    cell_size - each cell will be of size (cell_size, cell_size) pixels

    block_size - each block will be of size (block_size, block_size) cells

    nbins - number of histogram bins

Returns:
    bboxes - D x 4 bounding boxes that tell [xmin ymin width height] per bounding
        box

    scores - the SVM scores associated with each bounding box in bboxes

You can compute the HoG features using the compute_hog_features() method
that you implemented in PS3. We have provided an implementation in utils.py,
but feel free to use your own implementation. You will use the HoG features
in a sliding window based detection approach.

Recall that using a sliding window is to take a certain section (called the
window) of the image and compute a score for it. This window then "slides"
across the image, shifting by either n pixels up or down (where n is called
the window's stride).
```

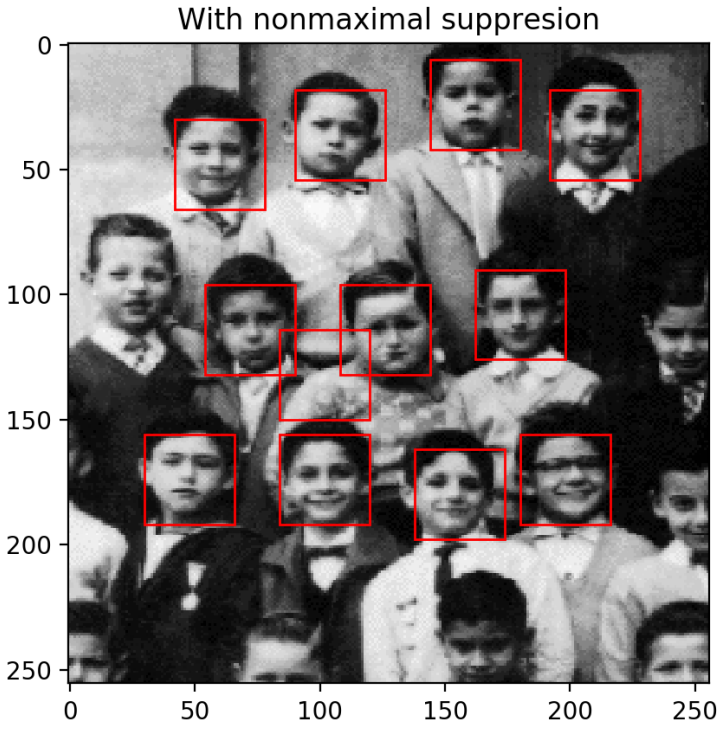
```
Using a sliding window approach (with stride of block_size * cell_size / 2),
compute the SVM score for that window. If it's greater than 1 (the SVM decision
boundary), add it to the bounding box list. At the very end, after implementing
nonmaximal suppression, you will filter the nonmaximal bounding boxes out.
'''
def run_detector(im, clf, window_size, cell_size, block_size, nbins, thresh=1):
    # initialize parameters
    im_h, im_w = im.shape[0], im.shape[1]
    window_h, window_w = window_size[0], window_size[1]
    stride = block_size * cell_size / 2

    # sliding windows
    bboxes = []
    scores = []

    for i in range(0, im_w - window_w, stride):
        for j in range(0, im_h - window_h, stride):
            bbox = [i, j, window_w, window_h]
            im_i = im[j:j+window_h, i:i+window_w]
            features_i = compute_hog_features(im_i, cell_size, block_size, nbins)
            score_i = clf.decision_function(features_i.flatten().reshape(1, -1))
            if score_i > thresh:
                bboxes.append(bbox)
                scores.append(score_i)

    # reshape it to be a numpy array
    bboxes = np.array(bboxes)
    scores = np.array(scores)
    print bboxes
    print
    print scores
    return bboxes, scores

'''
NON_MAX_SUPPRESSION Given a list of bounding boxes, returns a subset that
uses high confidence detections to suppresses other overlapping
detections. Detections can partially overlap, but the
center of one detection can not be within another detection.

Arguments:
    bboxes - ndarray of size (N,4) where N is the number of detections,
        and each row is [x_min, y_min, width, height]

    confidences - ndarray of size (N, 1) of the SVM confidence of each bounding
        box.
```

```
    img_size - [height,width] dimensions of the image.

Returns:
    nms_bboxes -  ndarray of size (N, 4) where N is the number of non-overlapping
        detections, and each row is [x_min, y_min, width, height]. Each bounding box
        should not be overlapping significantly with any other bounding box.

In order to get the list of maximal bounding boxes, first sort bboxes by
confidences. Then go through each of the bboxes in order, adding them to
the list if they do not significantly overlap with any already in the list.
A significant overlap is if the center of one bbox is in the other bbox.
'''
def non_max_suppression(bboxes, confidences):
    nms_bboxes = []
    indices = np.argsort(-confidences.reshape(1, -1)).flatten()

    for i in xrange(indices.shape[0]):
        bbox = bboxes[indices[i], :]
        if i == 0:
            nms_bboxes.append(bbox)
        else:
            isValid = True
            xmin, ymin, w, h = bbox[0], bbox[1], bbox[2], bbox[3]
            xc = (xmin + (xmin+w)) / 2.0
            yc = (ymin + (ymin+h)) / 2.0

            for j in xrange(len(nms_bboxes)):
                _xmin, _ymin, _w, _h = nms_bboxes[j][0], nms_bboxes[j][1],
                                       nms_bboxes[j][2], nms_bboxes[j][3]
                _xmax, _ymax = (_xmin + _w), (_ymin + _h)
                if (_xmin <= xc <= _xmax) and (_ymin <= yc <= _ymax):
                    isValid = False
                    break

            if isValid:
                nms_bboxes.append(bbox)

    nms_bboxes = np.array(nms_bboxes)
    return nms_bboxes
```

(a) The output image is shown below:



(b) The output image is shown below:



2 Image Segmentation

The following code includes implementations of `kmeans_segmentation()` and `meanshift_segmentation()`:

```
'''
KMEANS_SEGMENTATION: Image segmentation using kmeans
Arguments:
    im - the image being segmented, given as a (H, W, 3) ndarray

    features - ndarray of size (#pixels, M) that are the feature vectors
        associated with each pixel. The #pixels are arranged in such a way
        that calling reshape((H,W)) will correspond to the image im.

    num_clusters - The parameter "K" in K-means that tells the number of
        clusters we will be using.

Returns:
    pixel_clusters - H by W matrix where each index tells what cluster the
        pixel belongs to. The clusters must range from 0 to N-1, where N is
        the total number of clusters.

The K-means algorithm can be done in the following steps:
(1) Randomly choose the initial centroids from the features
(2) Repeat until convergence:
    - Assign each feature vector to its nearest centroid
    - Compute the new centroids as the average of all features assigned to it
    - Convergence happens when the centroids do not change
'''
def kmeans_segmentation(im, features, num_clusters):
    # initialize some parameters
    pixel_num = features.shape[0]
    centeroid_indices = np.random.randint(pixel_num, size=num_clusters)
    # randomly choose the initial centroids
    current_centeroids = features[centeroid_indices]

    # repeat until convergence
    while True:
        # find the distances of one pixel relative to all centeroids
        dist = np.zeros((pixel_num, num_clusters))
        for i in xrange(num_clusters):
            dist[:, i] = np.linalg.norm(features - current_centeroids[i, :], axis=1)

        # find the nearest cluster
        nearest_clusters = np.argmin(dist, axis=1)
        # update centeroids
        prev_centeroids = current_centeroids
```

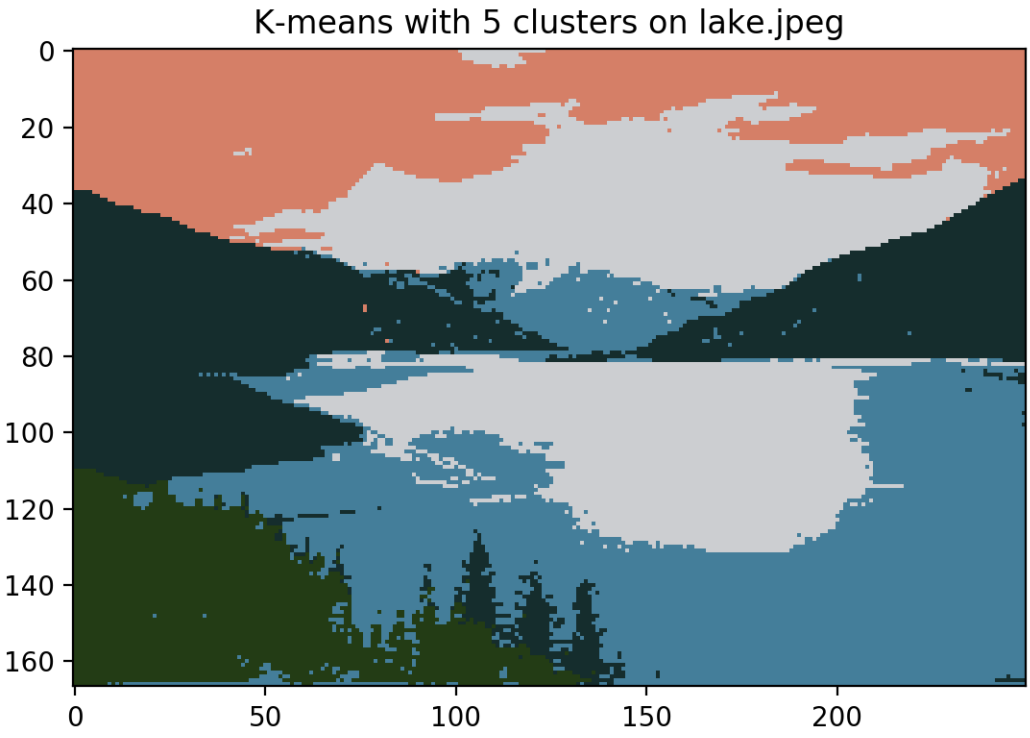
```
        for i in xrange(num_clusters):
            pixel_index = np.where(nearest_clusters == i)
            cluster_features = features[pixel_index]
            current_centeroids[i, :] = np.mean(cluster_features, axis=0)

        # break when converged
        if np.array_equal(current_centeroids, prev_centeroids):
            break

    pixel_clusters = np.reshape(nearest_clusters, (im.shape[0], im.shape[1]))
    return pixel_clusters


'''
MEANSHIFT_SEGMENTATION: Image segmentation using meanshift
Arguments:
    im - the image being segmented, given as a (H, W, 3) ndarray

    features - ndarray of size (#pixels, M) that are the feature vectors
        associated with each pixel. The #pixels are arranged in such a way
        that calling reshape((H,W)) will correspond to the image im.

    bandwidth - A parameter that determines the radius of what participates
       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 0 to N-1, 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. The output of this step will be a mean vector.
        For each iteration of the meanshift, if another pixel is within the
        bandwidth circle (in feature space), then that pixel should also be
        marked as seen
    - If the output mean vector from the mean shift step is
        sufficiently close (within half a bandwidth) to another cluster
        center, say it's part of that cluster
    - If it's not sufficiently close to any other cluster center, make
        a new cluster
```

```
    (3) After finding all clusters, assign every pixel to the nearest cluster
    in feature space.

    To perform mean shift:
        - Once a random pixel has been selected, pretend it is the current mean
          vector.
        - Find the feature vectors of the other pixels that are within the
          bandwidth distance from the mean feature vector according to EUCLIDEAN
          distance (in feature space).
        - Compute the mean feature vector among all feature vectors within the
          bandwidth.
        - Repeat until convergence, using the newly computed mean feature vector
          as the current mean feature vector.
    '''
    def meanshift_segmentation(im, features, bandwidth):
        # initialize some parameters
        H, W = im.shape[0], im.shape[1]
        pixel_num, M = features.shape
        mask = np.ones(pixel_num)
        clusters = []

        while np.sum(mask) > 0:
            loc = np.argwhere(mask > 0)
            idx = loc[int(np.random.choice(loc.shape[0], 1)[0])][0]
            mask[idx] = 0

            current_mean = features[idx]
            prev_mean = current_mean

            while True:
                dist = np.linalg.norm(features - prev_mean, axis=1)
                incircle = dist < bandwidth
                mask[incircle] = 0
                # update current_mean
                current_mean = np.mean(features[incircle], axis=0)
                if np.linalg.norm(current_mean - prev_mean) < 0.01 * bandwidth:
                    break
                prev_mean = current_mean

            isValid = True
            for cluster in clusters:
                if np.linalg.norm(cluster - current_mean) < 0.5 * bandwidth:
                    isValid = False
            if isValid:
                clusters.append(current_mean)
```

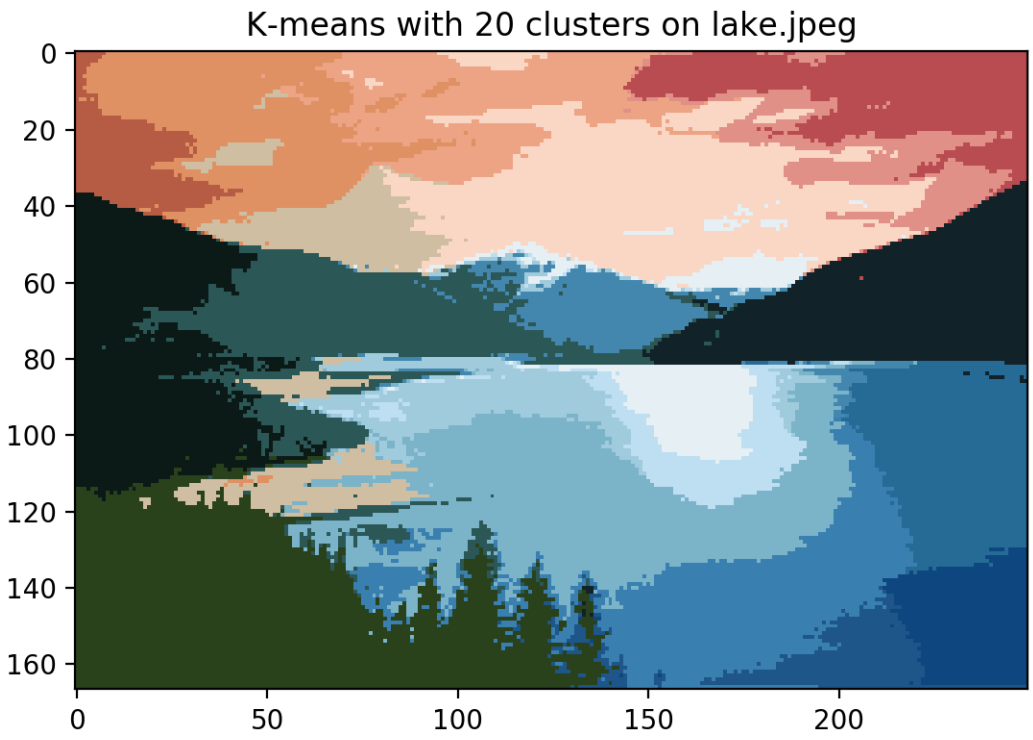
```
pixel_clusters = np.zeros((H, W))
clusters = np.array(clusters)
for i in range(pixel_num):
    idx = np.argmin(np.linalg.norm(features[i, :] - clusters, axis=1))
    pixel_clusters[i / W, i % W] = idx
return pixel_clusters.astype(int)
```

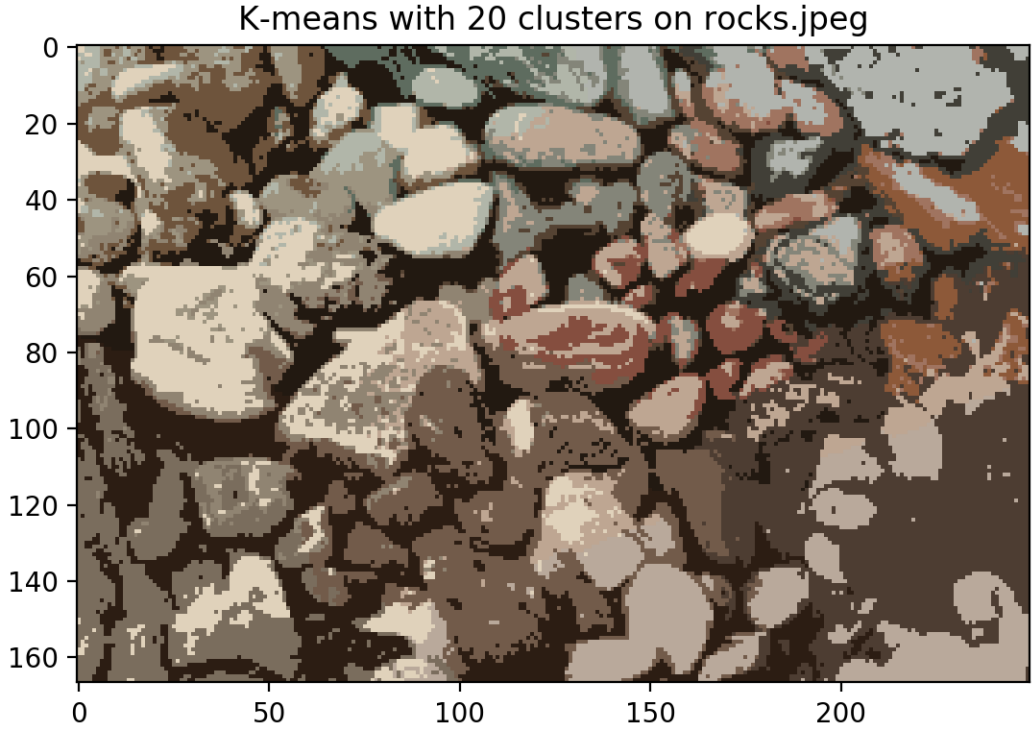
K-means

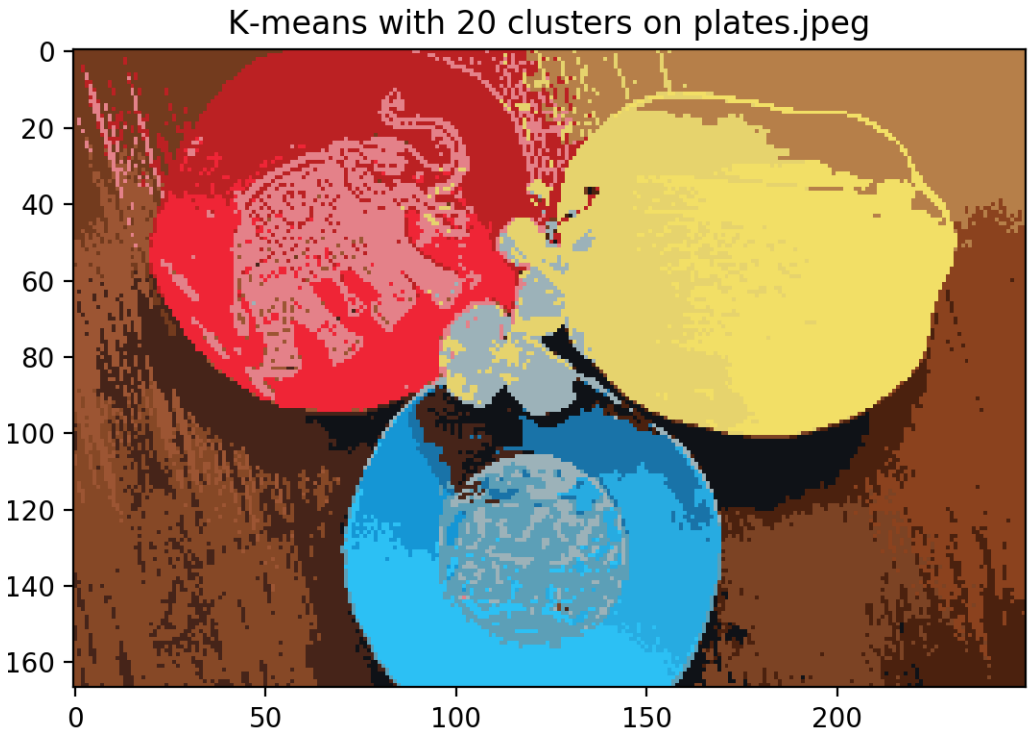
(a) The resulting segmentation on `lake.jpg` with default parameter is shown below:



(b) The default resulting segmentation works fine, but by increasing the number of clusters, much better results can be obtained. The number of clusters represents the ability of segmenting details of K-means algorithm, *i.e.* "resolution" of segmentation. The greater the number of clusters, the better segmentation results are. The following images show the results on `lake.jpg`, `rock.jpg` and `plates.jpg` with 20 clusters and one can easily see finer details in images.

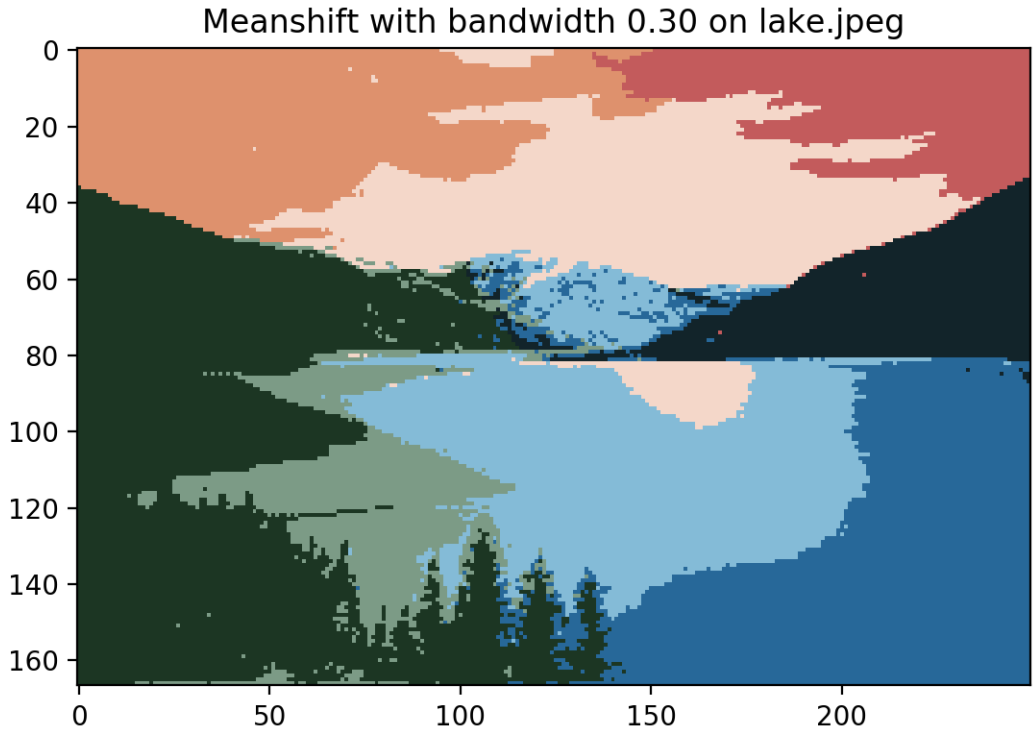

K-means with 20 clusters on lake.jpeg
0
20
40
60
80
100
120
140
160
0
50
100
150
200


K-means with 20 clusters on rocks.jpeg
0
20
40
60
80
100
120
140
160
0
50
100
150
200



Meanshift

(a) The resulting segmentation on `lake.jpg` with default parameter is shown below:



(b) The segmentation results on `lake.jpg` and `lake.jpg` with default parameter are already pretty good: