ECE 571 Project 3
Color Image Compression Using Unsupervised Learning

Emily Herron
eherron5@vols.utk.edu

April 2, 2019

Contents

1 Abstract

This report examines the results obtained from applying four clustering methods to the color values of an RGB image in order to optimally represent it with a smaller number of colors. Before conducting each experiment, the image is converted to a dataset of feature vectors corresponding to color values. The K-means, Winner-take-all, and Kohonen feature maps clustering algorithms are applied to the dataset and the results are used to render the image using only 2, 4, 8, 16, 32, 64, 128, and 256 colors. The Mean shift clustering algorithm is also used to determine the optimal clusters for the image data. By comparing the mean squared errors of each result, it is concluded that the K-means clustering algorithm produces the best overall representation of the original image.

2 Introduction

2.1 Topic Overview

Many problems in the field of Pattern Recognition involve working with unlabeled datasets. Supervised learning approaches such as the Maximum A-Posteriori Probability and K-Nearest Neighbor classifiers cannot utilized in such cases. Instead, unsupervised learning methods, which derive knowledge from these datasets by determining the relationships between samples based on their features, are appropriate to these problems [2]. Clustering algorithms, which identify homogeneous subgroups or "clusters" given a set of samples, make up a large subset of unsupervised learning methods [4]. Many of these methods perform clustering by computing distances between each data sample and a set of cluster centroids or a set of nearby samples. Some examples of commonly-used clustering algorithms include K-Means, Fuzzy C-means, Hierarchical clustering, and Gaussian mixtures [1].

2.2 Objective

The central objective of this project is to apply a set of clustering methods to a dataset containing an image's red, green, and blue (RGB) pixel components in order to find the 256 or fewer colors that best represent the original image. The K-Means, Winner-take-all, and SOM algorithms have been implemented and used to group samples corresponding to RGB color values for each pixel of the image 'flowersm.ppm' into a specified number of clusters. The samples are assigned new RGB values based on the coordinate of each assigned cluster's centroid. The Mean Shift algorithm is also used to determine optimal color cluster assignments for a set of window sizes. The outcomes of each approach have been evaluated using mean squared error and the results are given and discussed in this report.

3 Technical Approach

3.1 Clustering

3.1.1 K-Means Clustering

K-means clustering is one of the most popular clustering algorithms. It works by alternating between assigning data samples to clusters determined by a set of k arbitrary centroids and adjusting the centroids to fit the updated clusters. The algorithm proceeds according to the following steps:

1. Initialize a set of k arbitrary cluster centers, $\omega_1, \omega_2 ... \omega_k$.
2. Assign each sample in the dataset$\vec{x}$ to the cluster having the center with the smallest Euclidean distance to the sample.
3. Update the center or mean of each cluster.
4. Repeat steps 2 and 3 until cluster assignments are unchanged or until the maximum number of iterations is exceeded [3].

3.1.2 Winner-Take-All Approach

The Winner-take-all approach works similarly to K-means. However, this algorithm adjusts the centroids at each iteration by moving them in the direction of the samples that are closest to them. The steps are as follows:

1. Initialize a set of k arbitrary cluster centers, $\omega_1, \omega_2 ... \omega_k$.
2. For each sample $\vec{x}$ find the cluster center ω_α with the closest Euclidean distance.
3. Update ω_α using the formula $\omega_\alpha^{new} = \omega_\alpha^{old} + \epsilon(\vec{x} - \omega_\alpha^{old})$where ϵ is a small learning parameter.
4. Repeat steps 2 and 3 until cluster assignments are unchanged or until the maximum number of iterations is exceeded [3].

3.1.3 Kohonen Maps

The Kohonen or self-organizing maps approach is an extension of Winner -take-all. In both methods, the coordinates of the "winning" centroids closest to each sample are updated in every iteration. However, with Kohonen feature maps, a problem-dependent topological distance is assumed between each centroid, which is in this implementation the Euclidean distance between the RGB values of each centroid. As each winning centroid is updated, its neighbors are also updated and shifted in the direction of each sample. The steps below summarize this algorithm.

1. Initialize a set of k arbitrary cluster centers, $\omega_1, \omega_2 ... \omega_k$.

2. For each sample $\vec{x}$ find the cluster center ω_{winner} with the closest Euclidean distance, as well as all centroids, w_r, falling within a given radius, denoted by σ.

3. Update each cluster and its neighbors, ω_r, using the formula

$$\omega_r^{t+1} = \omega_r^{t+1} + \epsilon(k)\phi(k)(\vec{x} - \omega_r^t)$$

   Note that

$$\epsilon(k) = \epsilon_{max}(\frac{\epsilon_{min}}{\epsilon_{max}})^{\frac{t}{t_{max}}}$$

   where t is the current time step or iteration, t_{max} is the maximum number of iterations, and ϵ_{min} and ϵ_{max} define the minimum and maximum learning rates. Furthermore,

$$\phi(k) = -\frac{||g_{w_r} - g_{w_{winner}}||^2}{2\sigma^2}$$

   where g_{w_r} and $g_{w_{winner}}$ are the coordinates of the neighboring and winning clusters.

4. Repeat steps 2 and 3 until cluster assignments are unchanged or until the maximum number of iterations is exceeded [3].

3.1.4 Mean-Shift Clustering

Unlike the previous clustering methods, mean-shift clustering does not assign samples to a fixed number of clusters. Instead, this non-parametric approach determines an optimal number of clusters by shifting a kernel or window of size h toward the mean of the samples enclosed in the window, as described by the following steps:

1. Apply the kernel function, $K(x)$ to each sample x:

$$K(x) = \begin{cases} 1 & \text{if } ||x|| \geq h, \\ 0 & \text{if } ||x|| < h \end{cases}$$

2. For each sample, x, find the mean, $m(s)$ of the data within the window centered at x such that

$$m(s) = \frac{\sigma_{x \in \omega_x} K(s - x)s}{\sigma_{x \in \omega_x} K(s - x)}$$

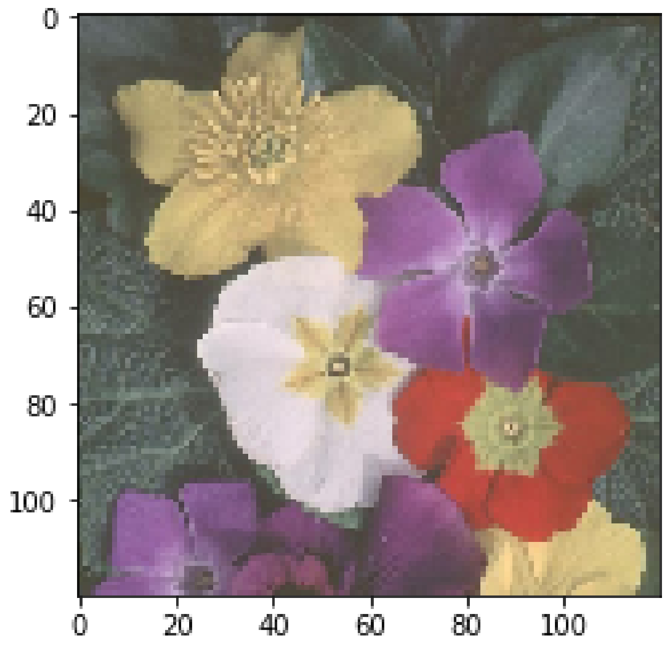


Figure 1: The original version of flowersm.ppm.

3. Shift the window to the mean by setting $x = m(s)$.
4. Repeat steps 2 and 3 if $||m(x) - x|| > \epsilon$ or until the maximum number of iterations is exceeded.

3.2 Performance Evaluation

Mean squared error is used to compare the RGB values of the images compressed in the clustering experiments to that of the original images. The error was computed based on the following function, where N is the total number of samples, x_i is the sample value prior to clustering, and $\tilde{x_i}$ is the sample after clustering.

$$MSE = \frac{1}{N}\Sigma_{i=1}^{N}(x_i - \tilde{x_i})^2$$

4 Experiments and Results

4.1 Image Preprocessing

RGB pixel values are obtained from the image used in these experiments, flowersm.ppm, via Python's Matplotlib Image library. This original image is shown in Figure 1. The library's imread function reads the image as a 120 by 120 matrix of RGB integer components with values ranging from 0 to 255. The matrix is flattened into an array of RGB features with shape 14400 by 3. The RGB values are not normalized for these experiments.

4.2 K-Means Clustering

The K-means clustering algorithm is applied to the RGB feature data. The results of representing the image in terms of colors given by 2, 4, 8, 16, 32, 64, 128, and 256 cluster centers is shown in Figure 2. Likewise, the mean squared errors between each compressed image and the original image are shown in Figure 3 and listed in Table 1.

Table 1: Mean squared errors of flowersm.ppm expressed in k colors by means of K-means clustering.

k	Mean Squared Error
2	740.22296
4	325.60699
8	172.25535
16	76.50451
32	50.43428
64	33.38375
128	22.82532
256	15.38755

By comparing the images shown in Figure 2 with the original image and by observing the error trends plotted in Figure 3, it may be noted that the MSE stabilizes after k-values of 16 or greater. This convergence at lower color totals could be explained by the relatively low color variety of the image or by the robustness of this mean-based clustering approach. These findings may also attest to the popularity of the K-means clustering algorithm.

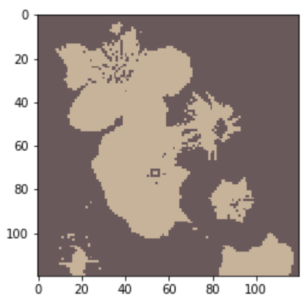
4.3 Winner-Take-All Clustering

Like K-Means, the Winner-take-all algorithm clusters the image's RBG values, mapping them to the given number of centroids. The images resulting from centroid totals of 2, 4, 8, 16, 32, 64, 128, and 256 at a learning rate, ϵ, of 0.01 are shown in Figure 4. The corresponding mean squared errors are plotted in Figure 5 and specified in Table 2.

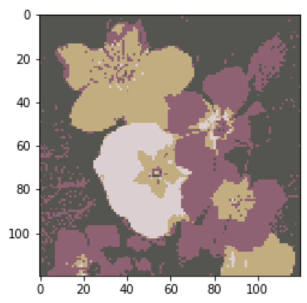
Table 2: Mean squared errors of flowersm.ppm expressed in k colors using the Winner-take-all approach with $\epsilon = 0.01$.

k	Mean Squared Error
2	824.63266
4	527.64150
8	293.88688
16	164.34924
32	79.15706
64	39.92840
128	31.68938
256	17.76625

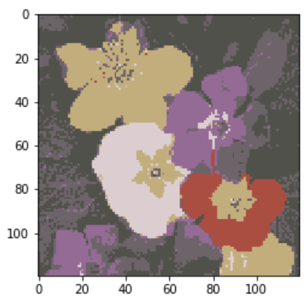
Note that in contrast to the results obtained using K-means clustering, the mean squared error



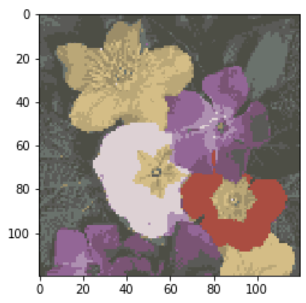
(a)



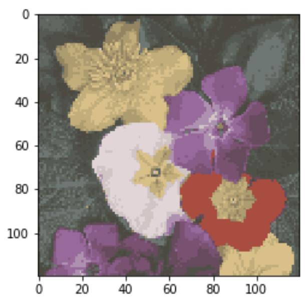
(b)



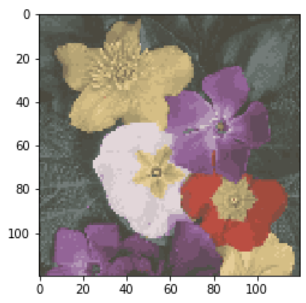
(c)



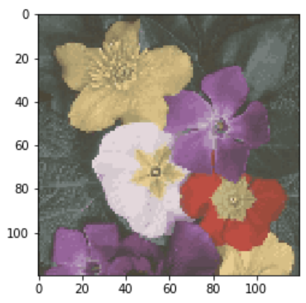
(d)



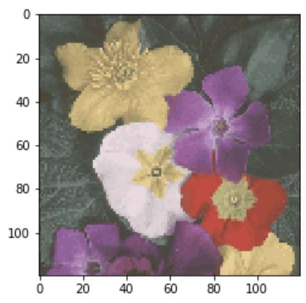
(e)



(f)



(g)



(h)

Figure 2: Images produced by applying K-means clustering to the RGB color values of flowersm.ppm. Subfigures (a)-(h) correspond to k values of 2, 4, 8, 16, 32, 64, 128, and 256.

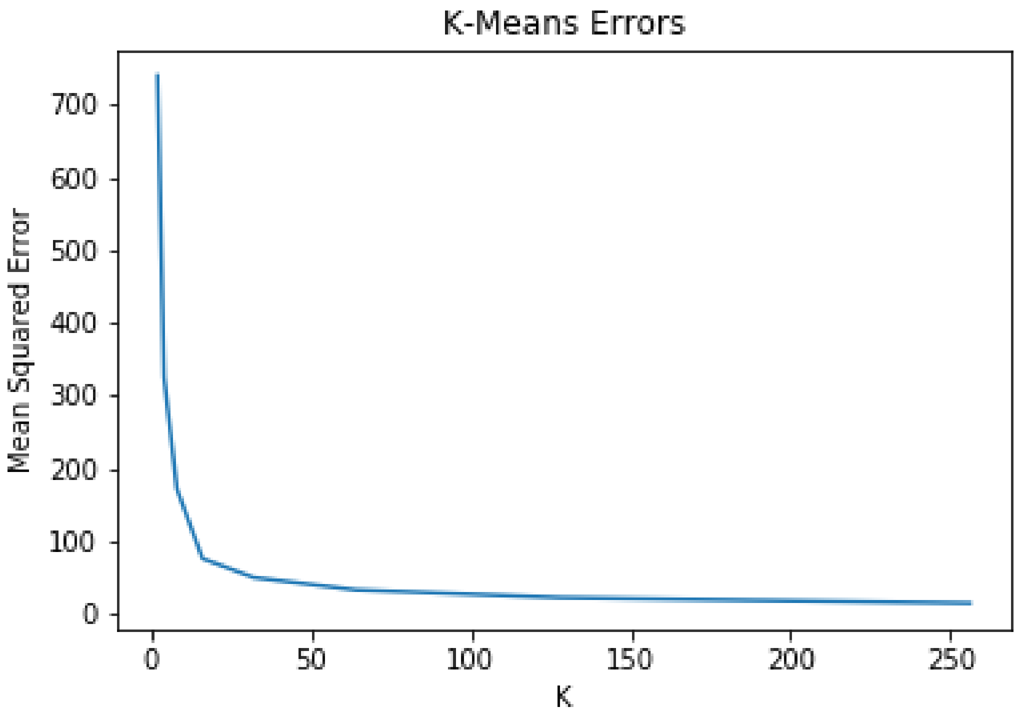


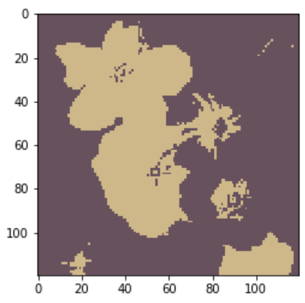
Figure 3: Mean squared errors of flowersm.ppm's color values expressed in k colors relative to those of the original image.

trends of images produced using this method do not begin to stabilize until color values of 64 or higher are used. This difference might be observed because the Winner-take-all approach works by repeatedly shifting cluster centroids towards each sample by a factor of some learning rate, rather than constantly updating the centroids to the means of each cluster. Given this explanation, it may be interesting to compare the mean squared errors obtained using different learning rates.

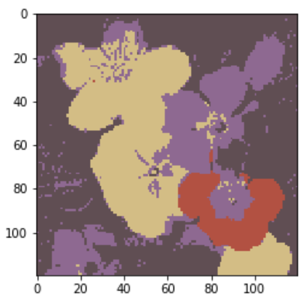
4.4 Kohonen Feature Maps Clustering

The Kohonen Maps clustering approach is also applied to the image data. The images compressed to 2, 4, 8, 16, 32, 64, 128, and 256 total colors using this method with $\sigma = 10$, $\epsilon_{min} = 0.0001$, and $\epsilon_{max} = 0.1$ are shown in Figure 6. The mean squared errors between each image's converted RGB values and those of the original are graphed in Figure 7 and listed in Table 3.

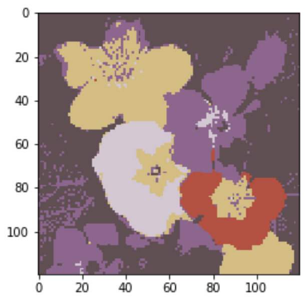
The trend in error outcomes for the compressed images obtained at each value of k shown in Figure 7 does not begin to stabilize before values of k greater than 128. This suggests that this clustering approach offers the least robust results and requires higher numbers of clusters to achieve more accurate results in comparison to the previous two methods. These results may be observed due to the repeated shifts of the neighboring centroids towards the winning centroid, which may be relatively distant from one another, depending on the value of σ. In the future, it may be interesting to the compare the results produced using different ϵ_{min}, ϵ_{max}, and σ values.



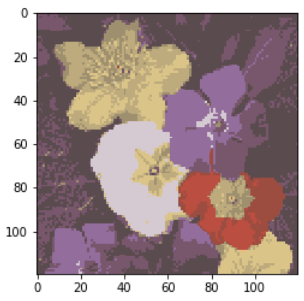
(a)



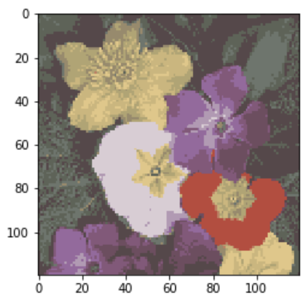
(b)



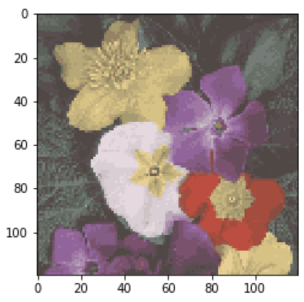
(c)



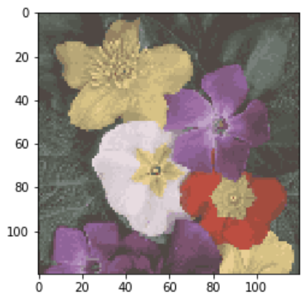
(d)



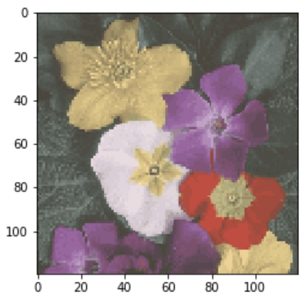
(e)



(f)



(g)



(h)

Figure 4: Images produced by applying the Winner-take-all algorithm, $\epsilon = 0.01$, to the RGB data of flowersm.ppm. Subfigures (a)-(h) correspond to centroid totals of 2, 4, 8, 16, 32, 64, 128, and 256.

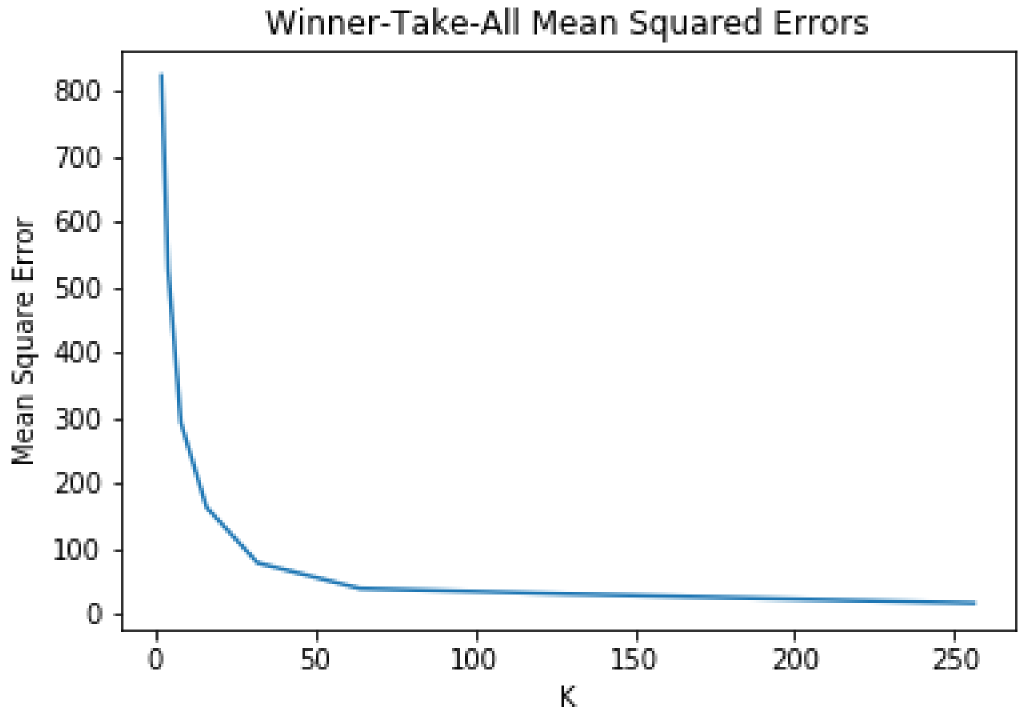


Figure 5: Mean squared errors of flowersm.ppm's color values expressed in k colors using the Winner-take-all approach with $\epsilon = 0.01$.

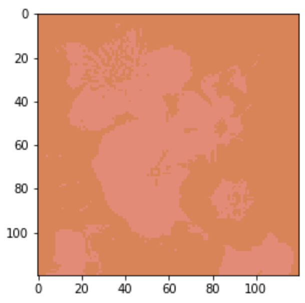
4.5 Mean Shift Clustering

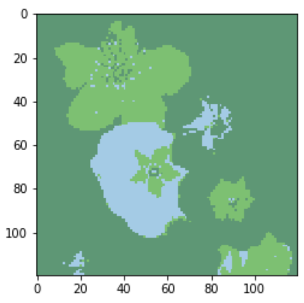
The images produced by applying Mean Shift clustering with window sizes, h =1, 2, 3, 4, and 5 with $\epsilon = 0.1$ are displayed in Figure 8. The mean squared errors of each converted image are also plotted in Figure 9.

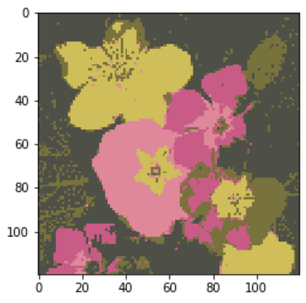
Note that the mean squared errors of the images transformed with window sizes of 3 or less are relatively low and similar and that the error drastically increases when h is greater than 3. This phenomenon may occur because a larger percentage of samples in the dataset fall within 4 or more units of eachother.

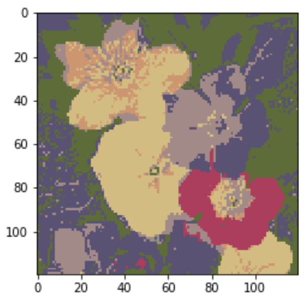
5 Summary and Future Work

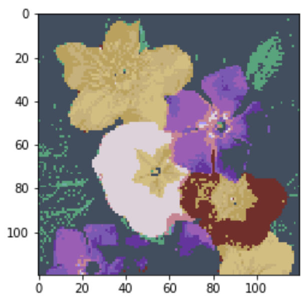
Four different clustering algorithms were implemented and applied to the color feature data of an image in order to represent the image using a lower number of colors. The performance of each classifier was evaluated both by inspecting each rendered compressed image and by determining the mean squared error of each relative to the original image. It was concluded that the K-means clustering algorithm achieved best overall performance when compared to the Winner-take-all and Kohonen maps approaches. K-means achieved the lowest resulting errors which stabilized when fewer total colors were used. The results of the Mean Shift clustering algorithm for different window

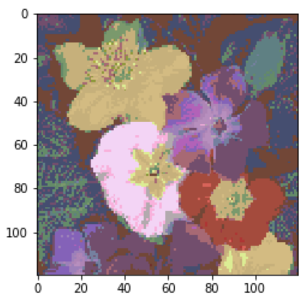

(a)

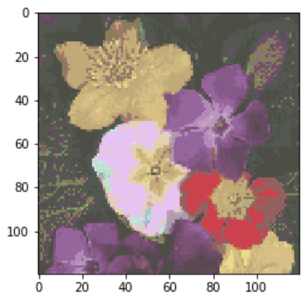

(b)


(c)


(d)


(e)


(f)


(g)

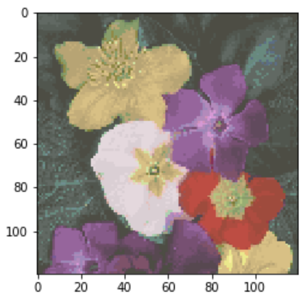

(h)

Figure 6: Images produced by applying the Kohonen feature maps clustering method with $\sigma = 10$, $\epsilon_{min} = 0.0001$, and $\epsilon_{max} = 0.1$ to the original image to express each image in terms of a specified number of colors. Subfigures (a)-(h) correspond to color totals of 2, 4, 8, 16, 32, 64, 128, and 256.

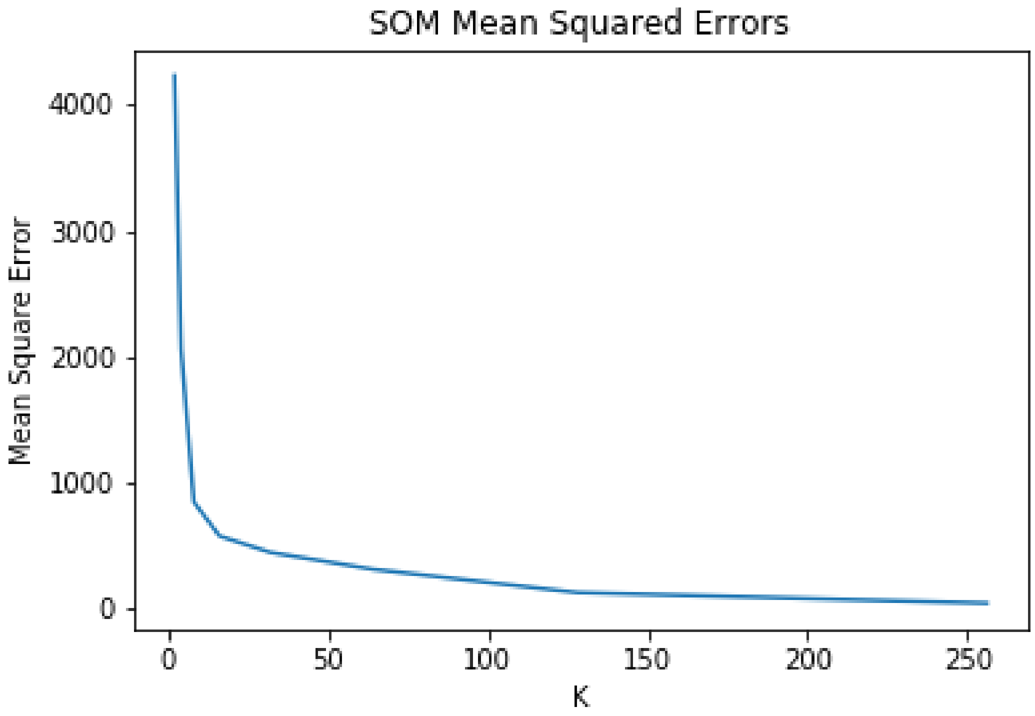


Figure 7: Mean squared errors of flowersm.ppm's RGB feature data when expressed in k colors using Kohonen feature maps with $\sigma = 10$, $\epsilon_{min} = 0.0001$, and $\epsilon_{max} = 0.1$.

sizes were also compared.

Some of the clustering methods, especially Kohonen feature maps clustering, were a challenge to conceptualize and implement in the beginning. However, through trial and error and hints from online tutorials, each was completed without too much difficulty. Through writing functions for each method and comparing the results achieved using different numbers of clusters, I have gained a stronger understanding of the differences in how each method works. To expand upon these results, additional experiments could be carried out to determine the effects of varying other parameters, such as the learning rates used in the Winner-take-all and Kohonen map approaches. The run times of each might also be recorded and compared.

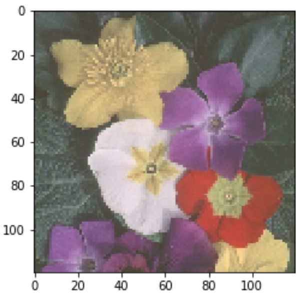
6 References

1. "A Tutorial on Clustering Algorithms", Politecnico di Milano, https://home.deib.polimi.it/matteucc/Clustering/

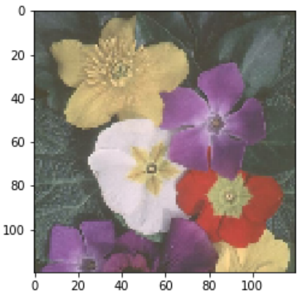
2. Chris Piech and Andrew Ng, "K Means", Stanford CS221, Stanford University, 2013, http://stanford.edu/ cpiech/

3. Hairong Qi, "Unsupervised Learning (Clustering)", Lecture, ECE471/571, University of Tennessee, Knoxville, Spring, 2019.

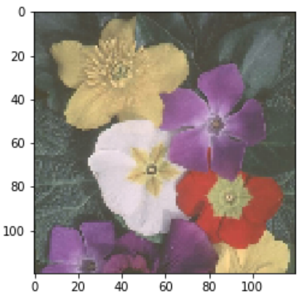
4. "Unsupervised Learning," Stanford University, https://lagunita.stanford.edu/c4x/HumanitiesScience/StatLearni



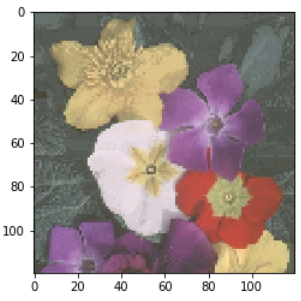
(a)



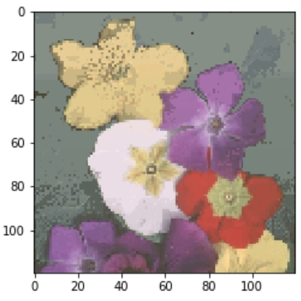
(b)



(c)



(d)



(e)

Figure 8: Images produced by applying the Mean Shift clustering method with $\epsilon = 0.1$ to the color data of flowersm.ppm. Subfigures (a)-(e) correspond to h values of 1, 2, 3, 4, and 5.

Table 3: Mean squared errors of flowersm.ppm expressed in k colors via Kohonen feature maps clustering $\sigma = 10$, $\epsilon_{min} = 0.0001$, and $\epsilon_{max} = 0.1$.

k	Mean Squared Error
2	4229.10076
4	2059.58433
8	846.71100
16	576.91651
32	446.08063
64	312.59407
128	127.76345
256	45.17016

Table 4: Mean squared errors of flowersm.ppm feature data clustered using Mean-shift clustering with $\epsilon = 0.1$.

h	Mean Squared Error
1	0.00000
2	0.05042
3	1.06350
4	24.63796
5	63.11394

7 Appendix

The following are a few of the functions referenced in this report. See the attached iPython Notebook for full code implementations code used to complete this project.

7.1 Image to Features Function

```
# convert image to features
def img_to_features(img):


    IMG_DIM = 120
    r = img[:,:,0]
    g = img[:,:,1]
    b = img[:,:,2]


    features = []
    for i in range(IMG_DIM):
        for j in range(IMG_DIM):
            features.append([r[i,j], g[i,j], b[i,j]])
    features = np.array(features)


    return features
```

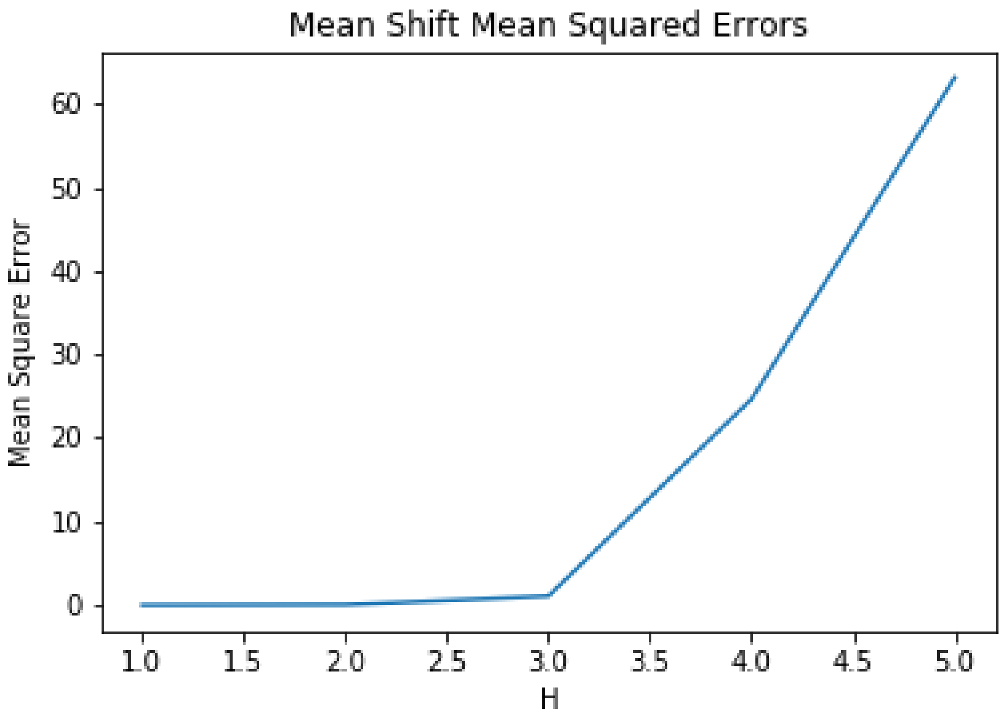


Figure 9: Mean squared errors of flowersm.ppm's RGB feature data when clustered using the Mean shift algorithm with $\epsilon = 0.1$.

7.2 Euclidean Distance Function

```
# function for calculating Euclidean distance
def euc_dist(x, y):

    diff = x-y

    if(diff.ndim == 1):
        return np.abs(diff)

    return np.linalg.norm(diff, axis = 1)
```

7.3 K-Means Clustering Function

```
# function for k-means clustering
def kmeans(X, k=256):

    # initialize means and indices
    prev_k_means = np.array([[0 for i in range(3)] for j in range(k)])
    means = np.array([[np.random.randint(INT_MIN, INT_MAX) for i in range(3)] for j in range(k)])
    indices = np.array([ind for ind in range(len(X))])

    count = 0
    nearest_means = []
    # continue iterations until cluster assignments stop changing or max iterations exceeded
    while not np.array_equal(prev_k_means, means) and count < 1000:
        count += 1
        nearest_means = []

        # find nearest centroid for each sample
        for x in X:
            s = euc_dist(means, x)
            nearest_means.append(np.argmin(s))
        nearest_means = np.array(nearest_means)
        prev_k_means = means
        means = np.array([[0,0,0] for i in range(k)])

        # update centroids to average of means in cluster
        for k_ in range(k):
            cluster = X[nearest_means[:]==k_]
            if len(cluster) == 0:
                means[k_] = prev_k_means[k_]
            else:
                means[k_] = np.array([np.mean(cluster, axis=0)])

    means = np.array([[int(np.round(means[j,i])) for i in range(3)] for j in range(k)])
    return nearest_means, means
```

7.4 Mean Squared Error Function

```
def calculate_error(features, nearest_means, means):
    new_features = []
    for i in range(len(nearest_means)):
        new_features.append(means[nearest_means[i]])
    new_features = np.array(new_features)
    return np.square(np.subtract(features, new_features)).mean()
```

7.5 Winner-Take-All Clustering Function

```
# winner take all algorithm
def winner_take_all(X, epsilon=0.01, k=256):

    #initialize centroids and indices
    prev_nearest_means = np.array([0 for j in range(len(X))])
    nearest_means = np.array([1 for j in range(len(X))])
    means = np.array([[np.random.uniform(INT_MIN, INT_MAX) for i in range(3)] for j in range(k)])
    indices = np.array([ind for ind in range(k)])


    count = 0
    # continue iterations until cluster assignments stop changing or max iterations exceeded
    while not np.array_equal(prev_nearest_means, nearest_means) and count < 1000:
        prev_nearest_means = np.copy(nearest_means)
        count += 1
        for i in range(len(X)):

            # find nearest centroid
            dists = euc_dist(means, X[i])
            s = np.column_stack((indices, dists))
            s = s[s[:,1].argsort()]
            cluster_ind = int(s[0,0])

            # use sample to update centroid and assign sample to cluster
            means[cluster_ind] = means[cluster_ind] + epsilon*(np.subtract(X[i], means[cluster_ind
            nearest_means[i] = cluster_ind

    means = np.array([[int(np.round(means[j,i])) for i in range(3)] for j in range(k)])
    return np.array(nearest_means), means
```

7.6 Kohonen Feature Map Clustering Function

```
def som(X, sigma=2.0, e_min=0.0001, e_max=0.15, k=256):
```

```
max_it = 100

prev_nearest_means = np.array([0 for j in range(len(X))])
nearest_means = np.array([1 for j in range(len(X))])
means = np.array([[np.random.uniform(INT_MIN, INT_MAX) for i in range(3)] for j in range(k)])
indices = np.array([ind for ind in range(k)])

# initialize means in grid formation
sqrt_k = int(np.ceil(np.sqrt(k)))
means = np.array(means)
plot_means(means, k, 'Before')

count = 0
while not np.array_equal(prev_nearest_means, nearest_means) and count < max_it:#000:
    prev_nearest_means = np.copy(nearest_means)

    count += 1
    print(count)
    for i in range(0, len(X)):

        dists = euc_dist(means, X[i])

        s = np.column_stack((indices, dists))
        s = s[s[:,1].argsort()]

        # get nearest cluster
        cluster_ind = int(s[0,0])

        s = s[s[:,1] <= sigma]

        cluster_i = np.floor(cluster_ind/sqrt_k)
        cluster_j = cluster_ind%sqrt_k

        e_k = e_max*(e_min/e_max)**(count/max_it)
```

```
            for j in range(len(s)):
                phi = np.exp(-np.linalg.norm(means[int(s[j,0])] - means[cluster_ind])**2/(2*sigma*
                means[int(s[j,0])] = means[int(s[j,0])] + e_k*phi*(np.subtract(X[i], means[cluster

            nearest_means[i] = cluster_ind

    means = np.array([[int(np.round(means[j,i])) for i in range(3)] for j in range(k)])
    plot_means(means, k, 'After')
    return np.array(nearest_means), means
```

7.7 Mean Shift Clustering Functions

```
# mean shift kernel function
def kernel(x, h):
    if np.linalg.norm(x) <= h:
        return 1.0
    else:
        return 0.0

# mean shift clustering function
def mean_shift(X, h, epsilon):

    X = np.copy(X)

    #X_old = np.array([[0.0 for i in range(3)] for j in range(len(X))])

    indices = np.array([float(ind) for ind in range(len(X))])

    # apply kernel function to all samples
    for i in range(0, len(X)):
        kernel(X[i], h)

    #X_old = np.copy(X)
    for i in range(0, len(X)):
```

```
    if i%1000 == 0:
        print('i', i)

    count = 0
    while True:

        count += 1

        dists = euc_dist(X, X[i])

        s = np.column_stack((indices, dists))

        #sort by distance
        s = s[s[:,1].argsort()]
        s = s[s[:,1] <= h]

        num = np.array([0.0, 0.0, 0.0])
        denom = 0.0

        for j in range(len(s)):
            num += np.multiply(kernel(X[int(s[j,0])]-X[i], h), X[int(s[j,0])])
            denom += kernel(X[int(s[j,0])]-X[i], h)

        # shift x
        mean = num/denom

        # save difference between mean and x
        diff = np.linalg.norm(mean-X[i])
        X[i] = mean

        # break if diff exceeds epsilon
        if diff < epsilon or count > 100:
            break
return X
```