Project 4 - Color Image Compression Using Unsupervised Learning (Clustering)

Elliot Greenlee
egreenle@vols.utk.edu

ECE 571 Pattern Recognition
April 21, 2017

Abstract

This report covers the design and implementation of various unsupervised clustering methods in order to perform image compression. K-means, winner-takes-all, Kohonen map, and mean-shift techniques are implemented. Compression is done on a multicolor RGB flower image. For each method, multiple compressions are attempted and evaluated. The results of each are compared visually to determine which method produces results most similar to the original. The mean-shift method was found to replicate the original most closely using 256 colors, and also took the least amount of time at 76.6 seconds.

1 Introduction

Humans make immediate discriminations between categories constantly throughout the day. These classifications are simple to a human, but would be difficult to implement on a computer. In order to train a program to complete such a task, statistical data must be input. Each data point is made up of mathematical features along with its class information. By analyzing this data a model can be built, reflective of the real world, that may be referenced later with new inputs for classification. [2]

The method of training using input data can take two forms: either supervised or unsupervised. In supervised training, the classes of training samples are known. This pre-classification is done by a human, a requirement that at the very least takes time, and in the extreme can be challenging. What happens when this human involvement is not tenable? This is where unsupervised classification is used. In unsupervised classification, the classes of samples are not known, and the number of appropriate classes must be determined. In order to solve these problems, clustering is used based on distance metrics. Between the features of samples is a calculated measure of similarity, such as Euclidean distance. This metric can be used to compare samples, and to find clusters of similar samples. In figure 1 from researcher Kadir Peker, samples are comprised of income and debt features. If the features are plotted, it is possible to see three separate clusters of individuals. In this case, the discrimination is easy. With more data, more dimensions, and closer clusters, the problem becomes more challenging, and more appropriate for a computer to solve.

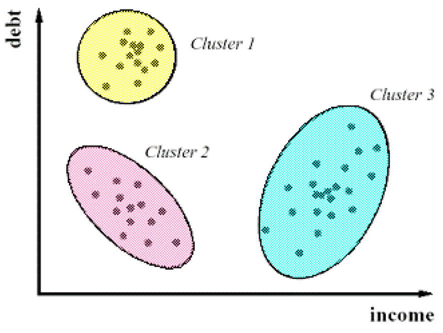

Figure 1: Income vs. debt cluster example

Unsupervised methods generally have the issue of not knowing how many separate classes are appropriate. Clustering methods can be separated into parametric and non-parametric groups. In parametric methods, the number of classes to attempt is input at the start of computation. This requires a brute force implementation on each number of classes, and can be time consuming on large sample sizes. In non-parametric methods, no prior knowledge is needed about the number of clusters. Instead, another measure of associativity might be used as a starting point for the guess. A low associativity threshold would telegraph a larger number of classes, while a higher associativity would indicate a smaller number of classes.

At the core of each clustering method is a measure of distance between two objects in the feature space. Generally, this is applied in three cases: between sample similarity, between cluster similarity, and sample-cluster similarity. In the simplest case of between sample similarity, typical Minkowski distances can be used without modification. When comparing to a cluster, a mean is calculated for the cluster and then that mean is treated as a point. Using this mean is called the centroid distance. Two other common possibilities for clusters are comparing the nearest and furthest neighbors. Typically, nearest neighbor leads to cluster structures reminiscent of minimum spanning tree, while furthest neighbor distance leads to highly interwoven clusters.

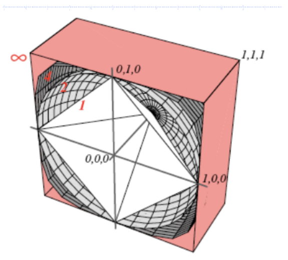

Figure 2: Various Minkowski distances

The objective of this project was to design and implement various clustering methods. Each of these methods was applied to color compression of an image, from 2^{24} to 2^8 colors in order to best represent the original image. This experiment gave an opportunity to explore clustering algorithms on a real problem, especially one that would be difficult for a human to determine. In this project, three parametric clustering algorithms are run on the compression problem under a range of class number assumptions. Additionally, a non-parametric method is used to automatically determine the number of classes. The overall effectiveness as well as the side effects of all methods are compared.

2 Technical Approach

The application of the various clustering techniques is on RGB image data. For this reason, samples and cluster centers are a three-dimensional Red, Green, Blue value.

2.1 K-Means

K-means is a centroid-based clustering technique, where clusters centers are represented by a structure identical to the individual samples. Because K-means is a parametric method, the number of clusters must be given at the start of calculation, a significant drawback. However, the simplicity of implementation and relatability to the k-nearest-neighbor algorithm makes this a commonly chosen method. The algorithm looks for the optimal cluster placement such that the squared distances to each sample in the cluster is minimized. The algorithm begins by creating arbitrary cluster centroids, and then assigning samples to the nearest centroid. The centroid is then recalculated as the mean of the assigned samples. Samples are reassigned, and if any sample classification has changed, then centroid is recalculated and the process begins again.

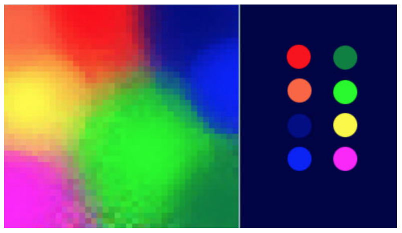

Figure 3: Representative clusters for an example image

2.2 Winner-Take-All

Winner-take-all is another centroid-based clustering technique similar to the K-means algorithm. The number of clusters must still be given at the start of calculation. In this method, a learning rate parameter is specified to control the movement of the centroid towards each sample. Each sample exerts a pull on the closest centroid during each iteration. The algorithm begins by creating arbitrary cluster centroids. Then for each sample, the closest centroid is chosen to be updated by adding the distance from the sample x to this winner ω_r using the formula

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

where ϵ is the learning rate, and is typically chosen to be on the order of 0.01.

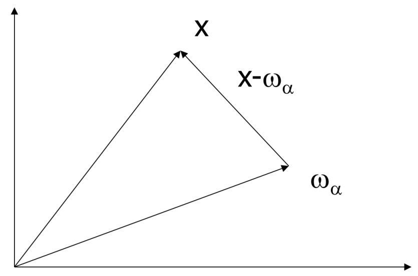

Figure 4: Winning cluster update for a sample in winner-takes-all

2.3 Kohonen Maps

Kohonen maps, or self-organizing maps, are similar to artificial neural networks in structure and are used for a type of dimensionality reduction. For each problem, a topological space is chosen on which to apply the cluster centers. The goal of this space is to cause different parts of the feature space to interact and improve performance. For each problem, a topological structure is assumed to exist between each pair of the cluster centers. This is not dependent on the clustering data in any way, and is purposefully assigned arbitrarily. Choices for this structure are single point, linear, and grid topologies. In the same way that the structure of an artificial neural network is arbitrary, but can affect the performance of training for a specific problem, so does the topology of the self-organizing map extend the winner-takes-all algorithm. As the winning cluster is updated, its neighbors in the topological space are also updated.

The algorithm for Kohonen maps runs in the same way as winner-takes-all. However, when a winner is chosen for a specific sample, that sample contributes to the movement of the winner and other clusters using the function

$$\omega_r^{k+1} = \omega_r^k + \epsilon * \Phi(k) * (x - \omega_r^k)$$

In this function, a learning rate ϵ is multiplied by a measure of topological distance $\Phi(k)$ from the winner and a feature distance $(x - \omega_r^k)$. The topological distance can be calculated using the function

$$\Phi = e^{\frac{\|gw_r - gw_{winner}\|^2}{2\sigma^2}}$$

. In our solution, clusters were randomly assigned to a linear value from one to the number of clusters, leading to a linear distance calculation for this topology. Other topologies are possible such as the grid topology shown below.

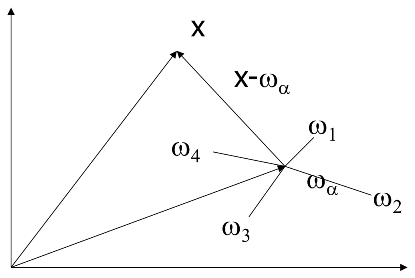

Figure 5: Winning cluster update for a sample in Kohonen mapping

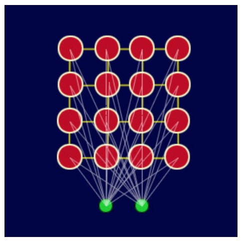

Figure 6: An example grid topology

2.4 Mean-Shift

Mean-shift, unlike the other methods explored, is non-parametric. No prior knowledge is needed abou the number of classes. Instead a key parameter, window size, is used to explore the feature space for clusters. A low window size causes a larger number of classes, while a higher window size leads to a smaller number of classes. This leads to challenges with determining a proper window size, and a slower convergence. Mean-shift is a density based clustering method, which means that clusters are represented as areas of higher density in the feature space. Objects trend towards the nearest cluster, which can be a varying shape, over time. The algorithm works by computing the mean of a window around each sample in the image using the equation

where K is the window function

$$K() =$$

This sample is then adjusted towards the mean. If the sample moves more than a specified tolerance value, then the process is repeated again on all samples. In the figures below, color clustering information for a house photo can be seen.

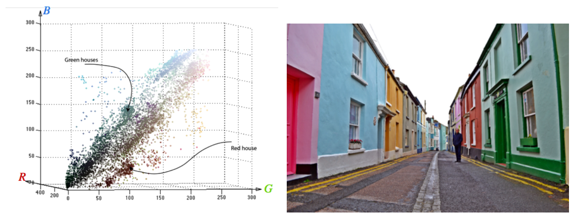

Figure 7: An example image and representative clustering in three dimensions.

3 Experiments and Results

3.1 Data

The data for this project is a 480x480 pixel full-color image (flowers.ppm). It is represented as RGB values at each coordinate of the picture. Each pixel of this color image has three components: red, green, and blue components. Each component is an 8-bit unsigned char. There are thus totally 2^{24} possible colors. In order to use the data, it was converted into a 480x480x3 matrix, a 3-dimensional feature space of all samples.

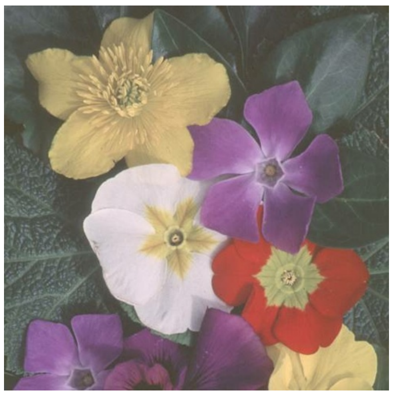

Figure 8: The original full color image

3.2 Clustering Reduction Performance

In this project, a full color image of 2^{24} possible colors is given, with the task of compressing the image to 256 (or 2^8) colors. By looking at the required compression, as well as some harsher compressions, we can get a better idea of the performance of each algorithm.

At harsher compression values, k-means is best at quickly producing the correct color for larger regions in the image. This comes at the expense of detail, which is slowly improved as the number of clusters increases. With just eight clusters, all the flower colors are correct.

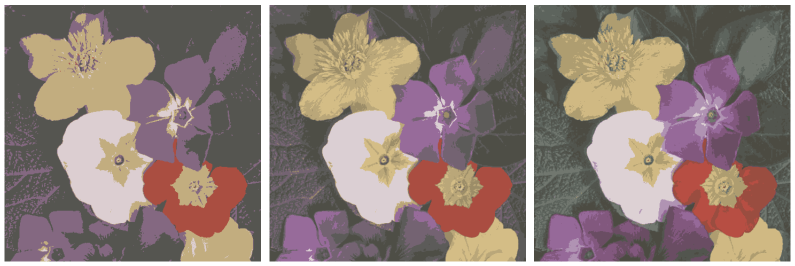

Figure 9: 8, 16, and 32 clusters using k-means

Winner-takes-all is the most balanced approach at lower cluster numbers, but struggles to really excel in any area. Image details are visible but not clear, and image colors are found, but not as quickly as in k-means.

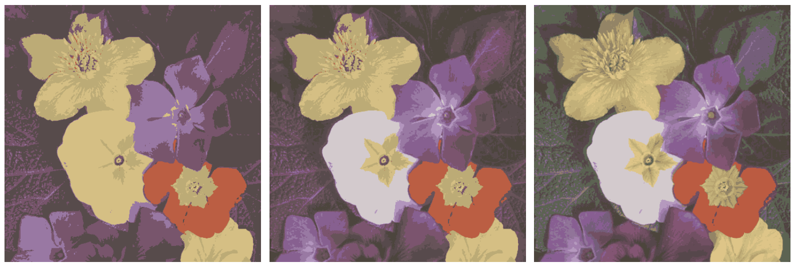

Figure 10: 8, 16, and 32 clusters using winner-taks-all

Kohonen maps was implemented using a linear topology. This seems to have produced a very rapid convergence on feature details, but struggles with proper coloring. In the figure, even at eight clusters the insides of the flowers are clear. However, at 32 clusters the red and white flowers are still purple and yellow.

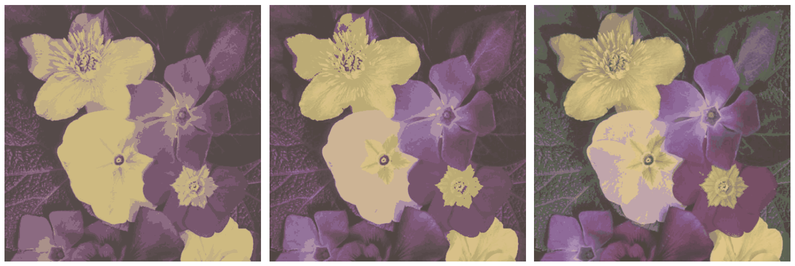

Figure 11: 8, 16, and 32 clusters using Kohonen maps

Mean shift was the best method at lower clustering values. In the two main metrics of color accuracy and detail, it had the best performance, quickly coloring all flowers correctly while preserving the internals of the flower. However, mean-shift suffered from random distortion patches of miscoloration, which is evident in the top left of the 11 cluster photo.

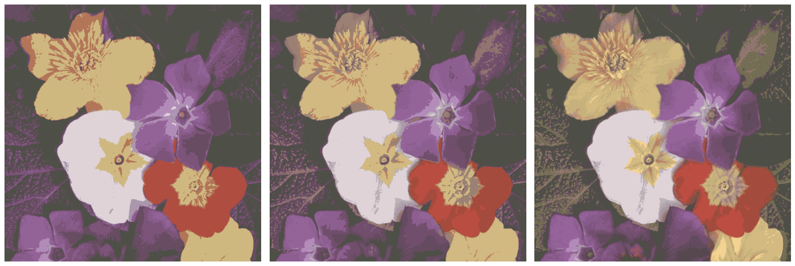

Figure 12: 11, 17, and 40 clusters using mean-shift

Of the four methods, mean-shift produced the best compression results at 256 clusters. While k-means and Kohonen maps produce the best results on the background leaves, mean-shift overall prevents patches of color, especially in regions like the details of the white flower. A close inspection shows few obscured details from the compression, and an overall high quality image.

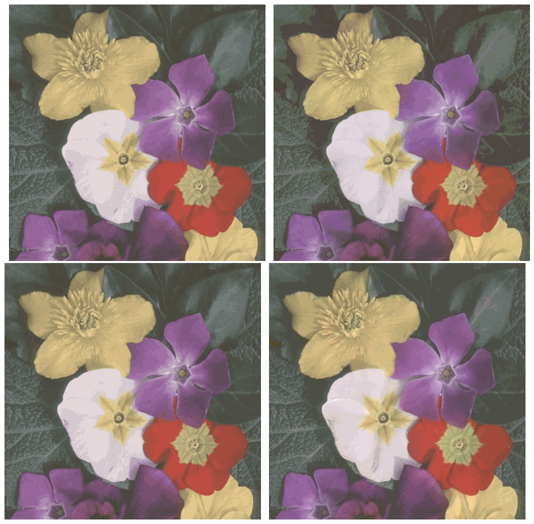

Figure 13: 256 clusters for k-means (TL), winner-takes-all (TR), Kohonen maps (BL), and mean-shift (BR)

3.3 Clustering Performance Costs

For each method, performance metrics were taken in order to determine the cost of compression. For k-means, winner-takes-all, and kohonen, the number of iterations to convergence were recorded, while the run timing were recorded for all methods. All runs were done on a Macbook Pro Retina 2.2 GHz Intel Core i7. In general across all methods, the average time needed per iteration doubled as the number of clusters doubled. This is in consensus with the theory, as each sample is compared to each cluster in every iteration. For the three parametric methods, the number of iterations steadily increased as about the same rate. The data for these methods is below.

Table 1. K-means iterations and average run time per iteration as clusters increase.

Clusters	Iterations	Avg. Time (s)
1	1	0.0
2	9	2.7
4	11	4.6
8	47	16.7
16	21	33.1
32	37	84.1
64	25	179.6
128	87	361.9
256	122	743.1

Table 2. Winner-takes-all iterations and average run time per iteration as clusters increase.

Clusters	Iterations	Avg. Time (s)
1	1	0.0
2	5	2.2
4	20	4.7
8	12	6.3
16	18	13.4
32	70	32.3
64	58	74.0
128	129	161.6
256	110	346.5

Table 3. Kohonen maps iterations and average run time per iteration as clusters increase.

Clusters	Iterations	Avg. Time (s)
1	1	0.0
2	4	4.1
4	7	9.2
8	27	20.4
16	41	41.1
32	102	83.0
64	95	166.4
128	200	320.1
256	130	702.5

Mean-shift was by far the fastest across all methods. This is because it was implemented using sci-kit learn, a Python package that is highly optimized for matrices and written in C. This is compared to a no-optimization Python implementation of the other three methods.

Table 4. Total time for all methods as clusters increased. Mean-shift clusters are slightly different and are indicated in parentheses. Times in seconds

Clusters	K-means	Winner-Takes-All	Kohonen Maps	Mean-Shift
1	0	0	0	N/A
2	24	11	16	N/A
4	51	94	64	10.4 (5)
8	785	76	551	28.9 (11)
16	695	241	1685	41.8 (17)
32	3112	2261	8466	53.6 (40)
64	4490	4292	15808	59.1 (100)
128	31485	20846	64020	62.7 (186)
256	90658	38115	91325	76.6 (348)

4 Summary

Four clustering methods: k-means, winner-takes-all, Kohonen maps, and mean-shift, were implemented. Each was applied to the problem of color compression on a single image. For each method, experimentation was done to highlight performance in relation to both image preservation and algorithm speed. Overall, mean-shift proved to be most successful at the required 256 compression. Additionally, mean-shift produced the fastest compression at 76.6 seconds. This is most likely a result of the underlying implementation in C, rather than a statement of the speed of mean-shift. In fact, typically mean-shift would produce the slowest convergence.

The objective of this project was to implement both parametric and non-parametric clustering algorithms, and then apply those algorithms to an image compression problem. I was able to more fully understand the practical implementations of these methods through trial and error, as well as the theory behind clustering techniques. Of special note was the realization that as a human

I would never be able to make the clustering determinations via eyesight that the computer did using the RGB feature space. Programming wise, I struggled with the run times of the various methods. However, this led to an adaptation on my part to a more distributed approach. In the future, threading would be very appropriate for this type of experimentation. It would be interesting to explore these compressions on various types of images in the future, to see what factors in the image contribute to the performance metrics.

References

[1] Duda, Richard O., Peter E. Hart, and David G. Stork. Pattern Classification. Second ed. New York: Wiley, 2001. Print.

[2] Ripley, B. D. Pattern Recognition and Neural Networks. N.p.: Cambridge University Press, 1996. Print.

[3] http://www.ai-junkie.com/ann/som/som1.html

5 Appendix

All code available at `https://github.com/LambentLight/flower-clustering`, which includes a small README. I am aware that the code copied and pasted here runs off of the page, but I was reluctant to change the formatting, which may obscure meaning. If you need the code, let me know at egreenle@vols.utk.edu and I will send it.

data.py

```
from scipy.misc import imread, imsave
import random
import math
import sys
import copy
import time

RED = 0
GREEN = 1
BLUE = 2


# clustering
class Unsupervised:
    def __init__(self, clusters, original_image):
        self.clusters = clusters
        self.original_image = original_image
        self.image = original_image.copy()
        self.image_width = len(original_image)
        self.image_height = len(original_image[0])

        self.classification = [[0 for x in range(self.image_width)] for y in ran
# every pixel location will have a classification from 0 to clusters-1

        # Generate original cluster centers
        self.cluster_centers = [[0 for x in range(3)] for y in range(self.cluste
# clusters x 3 vector with each randomly generated cluster center

        for center in self.cluster_centers:
            center[RED] = random.randint(0, 255)
            center[GREEN] = random.randint(0, 255)
            center[BLUE] = random.randint(0, 255)

    @staticmethod
    # TODO: is this math ok, especially for the kohonen?
    def euclidean(a, b):
        if len(a) != len(b):
            print("Euclidean math needs help")
            exit(1)

        sum = 0
        for i in range(len(a)):
            diff = a[i] - b[i]
            sum += math.pow(diff, 2)
        return math.sqrt(sum)

    # Assigns every image pixel to the closest center
    def assign_samples(self, distance):

        for pixel_i in range(self.image_height):
            for pixel_j in range(self.image_width):
                min_distance = sys.maxint
                min_cluster = -1
                for cluster, center in enumerate(self.cluster_centers):
                    new_distance = distance(center, self.image[pixel_i][pixel_j]
                    if new_distance < min_distance:
                        min_distance = new_distance
                        min_cluster = cluster

                self.classification[pixel_i][pixel_j] = min_cluster

    def different(self, old_classification, new_classification):
        for pixel_i in range(self.image_height):
            for pixel_j in range(self.image_width):
                if old_classification[pixel_i][pixel_j] != new_classification[pi
                    print "Difference found: ", pixel_i, pixel_j, old_classifica
                    return True

        return False

    # Compute the average difference between pixels in the original image and it
    def compression_error(self):
        diff = [0] * 3
        sum = 0
        for pixel_i in range(self.image_height):
            for pixel_j in range(self.image_width):
                diff[RED] = self.original_image[pixel_i][pixel_j][RED] - self.im
                diff[GREEN] = self.original_image[pixel_i][pixel_j][GREEN] - sel
                diff[BLUE] = self.original_image[pixel_i][pixel_j][BLUE] - self.

                diff_total = abs(diff[RED]) + abs(diff[GREEN]) + abs(diff[BLUE])

                sum += pow(diff_total, 2)

        average = sum / (self.image_width * self.image_height)

        return math.sqrt(average)

    def reduce_image(self, picture_name):
        print "Reducing Image"
        for pixel_i in range(self.image_height):
            for pixel_j in range(self.image_width):
                self.image[pixel_i][pixel_j] = self.cluster_centers[self.classifi
        imsave(picture_name, self.image)

    def kmeans_cluster(self):
        # store old classification
        old_classification = copy.deepcopy(self.classification)
        # assign samples
        self.assign_samples(self.euclidean)
        print "Initial assignment done"

        timing_sum = 0
        # while there are differences
        iteration = 1
        while self.different(old_classification, self.classification):
            start_time = time.time()
            print("Iteration: ", iteration)
            iteration += 1

            # store old classification
            old_classification = copy.deepcopy(self.classification)
            # calculate new cluster centers
            self.kmeans_centers()
            # assign samples
            self.assign_samples(self.euclidean)

            timing_sum += time.time() - start_time
            print time.time() - start_time

        timing_average = timing_sum / (iteration * 1.0)

        with open('kmeans_performance.txt', 'a') as f:
            f.write('{} {} {}\n'.format(self.clusters, iteration, timing_average

    def kmeans_centers(self):
        # vector of RGB clusters initialized to 0
        for center in self.cluster_centers:
            center[RED] = 0
            center[GREEN] = 0
            center[BLUE] = 0

        # iterate over pixels
        mean_counts = [0] * self.clusters
        for pixel_i in range(self.image_height):
            for pixel_j in range(self.image_width):
                cluster = self.classification[pixel_i][pixel_j]
                # add pixel values to that index in vector of RGB clusters
                mean_counts[cluster] += 1
                self.cluster_centers[cluster][RED] += self.image[pixel_i][pixel_
                self.cluster_centers[cluster][GREEN] += self.image[pixel_i][pixe
                self.cluster_centers[cluster][BLUE] += self.image[pixel_i][pixel

        # divide by number of samples in cluster
        for cluster, center in enumerate(self.cluster_centers):
            if mean_counts[cluster] == 0:
                mean_counts[cluster] = 1
            center[RED] /= mean_counts[cluster]
            center[GREEN] /= mean_counts[cluster]
            center[BLUE] /= mean_counts[cluster]

    def winner_cluster(self):
        # store old classification
        old_classification = copy.deepcopy(self.classification)
        # assign samples
        start_time = time.time()
        self.assign_samples(self.euclidean)
        print "Initial assignment done"

        timing_sum = 0
        # while there are differences
        iteration = 1
        while self.different(old_classification, self.classification):
            start_time = time.time()
            print("Iteration: ", iteration)
            iteration += 1

            # store old classification
            old_classification = copy.deepcopy(self.classification)
            # calculate new cluster centers
            self.winner_centers()
            # assign samples
            self.assign_samples(self.euclidean)

            timing_sum += time.time() - start_time
            time.time() - start_time

        timing_average = timing_sum / (iteration * 1.0)

        with open('winner_performance.txt', 'a') as f:
            f.write('{} {} {}\n'.format(self.clusters, iteration, timing_average
def winner_centers(self, learning_rate=0.01):
    # iterate over pixels
    for pixel_i in range(self.image_height):
        for pixel_j in range(self.image_width):
            cluster = self.classification[pixel_i][pixel_j]
            # add pixel values to that index in vector of RGB clusters
            self.cluster_centers[cluster][RED] += learning_rate * (self.imag
            self.cluster_centers[cluster][GREEN] += learning_rate * (self.im
            self.cluster_centers[cluster][BLUE] += learning_rate * (self.ima

def kohonen_cluster(self):
    # store old classification
    old_classification = copy.deepcopy(self.classification)
    # assign samples
    self.assign_samples(self.euclidean)
    print "Initial assignment done"

    timing_sum = 0
    # while there are differences
    iteration = 1
    while self.different(old_classification, self.classification):
        start_time = time.time()
        print("Iteration: ", iteration)
        iteration += 1

        # store old classification
        old_classification = copy.deepcopy(self.classification)
        # calculate new cluster centers
        self.kohonen_centers()
        # assign samples
        self.assign_samples(self.euclidean)

        timing_sum += time.time() - start_time
        time.time() - start_time

    timing_average = timing_sum / (iteration * 1.0)

    with open('kohonen_performance.txt', 'a') as f:
        f.write('{} {} {}\n'.format(self.clusters, iteration, timing_average

def kohonen_centers(self, learning_rate=0.01):
    # iterate over pixels
    for pixel_i in range(self.image_height):
        for pixel_j in range(self.image_width):
            winning_cluster = self.classification[pixel_i][pixel_j]
```

```
                for index, cluster in enumerate(self.cluster_centers):
                    cluster[RED] += learning_rate * self.closeness(winning_clust
                    cluster[GREEN] += learning_rate * self.closeness(winning_clu
                    cluster[BLUE] += learning_rate * self.closeness(winning_clus

    def closeness(self, winning_cluster, other_cluster, variance=1.0):
        return math.exp((-1.0 * pow(self.topological_distance(winning_cluster, o

    def topological_distance(self, winning_cluster, other_cluster):
        return winning_cluster - other_cluster

# Read in the image
original_image = imread('flowers.ppm')

# K-means
'''
for i in range(0, 9):
    clusters = pow(2, i)
    print "K-means, {} clusters".format(clusters)
    # kmeans = Unsupervised(clusters, copy.deepcopy(original_image))
    kmeans.kmeans_cluster()
    kmeans.reduce_image('images/k-means{}.ppm'.format(clusters))
'''

# Winner take all
'''
for i in range(0, 9):
    clusters = pow(2, i)
    print "Winner, {} clusters".format(clusters)
    winner = Unsupervised(clusters, copy.deepcopy(original_image))
    winner.winner_cluster()
    winner.reduce_image('images/winner{}.ppm'.format(clusters))
'''

# Kohonen
'''
for i in range(0, 9):
    clusters = pow(2, i)
    print "Kohonen, {} clusters".format(clusters)
    kohonen = Unsupervised(clusters, copy.deepcopy(original_image))
    kohonen.kohonen_cluster()
    kohonen.reduce_image('images/kohonen{}.ppm'.format(clusters))
'''

# Mean shift
import numpy as np
```

```
from sklearn.cluster import MeanShift

window_sizes = [11, 12, 13, 14]
for window_size in window_sizes:
    print "Mean Shift, window size {}".format(window_size)
    ms = MeanShift(bandwidth=window_size, bin_seeding=True)

    copy_image = copy.deepcopy(original_image)

    X = np.zeros((480*480, 3))
    # iterate over pixels
    for pixel_i in range(480):
        for pixel_j in range(480):
            X[pixel_i * 480 + pixel_j][RED] = copy_image[pixel_i][pixel_j][RED]
            X[pixel_i * 480 + pixel_j][GREEN] = copy_image[pixel_i][pixel_j][GRE
            X[pixel_i * 480 + pixel_j][BLUE] = copy_image[pixel_i][pixel_j][BLUE

    print X

    start_time = time.time()
    ms.fit(X)

    labels = ms.labels_
    print labels
    cluster_centers = ms.cluster_centers_
    with open('means_performance.txt', 'a') as f:
        f.write('{} {} {}\n'.format(window_size, time.time() - start_time, len(c
    print("number of estimated clusters : {}".format(len(cluster_centers)))
    print cluster_centers

    print "Reducing Image"
    for pixel_i in range(480):
        for pixel_j in range(480):
            copy_image[pixel_i][pixel_j][RED] = cluster_centers[labels[pixel_i *
            copy_image[pixel_i][pixel_j][GREEN] = cluster_centers[labels[pixel_i
            copy_image[pixel_i][pixel_j][BLUE] = cluster_centers[labels[pixel_i
    imsave('mean_shift{}.ppm'.format(window_size), copy_image)
```