Color Image Compression Using Unsupervised Learning

Kevin De Angeli
kevindeangeli@utk.edu
COSC 522 - Machine Learning
University of Tennessee, Knoxville

November 21, 2019

Abstract

This project explores unsupervised learning through the application of four clustering techniques. I wrote three classes that perform three popular algorithms: k-means, Winner-takes-all, and Kohonen Maps (also called Self-Organizing Maps). These algorithms were all written in Python with only Numpy. I have also used the Sklearn library to implement the mean-shift algorithm. The data set used in this project is just a simple picture of flowers with a lot of colors. I have used two metrics to illustrate performance: Mean Square Error and Peak Signal to Noise Ratio (PSNR). Overall, K-means seems to be the fastest and most accurate cluster algorithm for the purpose of image compression.

1 Experiments and Results

In general, cluster algorithms are divided into 2 groups: partitional and hierarchical. [1]. Partitional algorithms tend to divide the data into non-overlapping groups; this include algorithms suck as K-means [1]. On the other hand, hierarchical algorithms "use the distance matrix as input and create a hierarchical set of clusters" [1]. The code in the Appendix contains the three classes that I have written for this project. I have calculated the Mean Square Error (MSE) with Equation 1, and PSNR is computed with Equation 2. In general, as the MSE decreases, the PSNR should increase since they have an inverse relationship. This is illustrated in the comparison tables provided for each algorithm.

$$\frac{1}{n}\sum_{i=1}^{n}(y-\hat{y})^2 \tag{1}$$

$$20 * log_{10}(\frac{225}{\sqrt{MSE}}) \tag{2}$$

1.1 K-means

The first cluster algorithm that I applied for image compression is K-means. The steps of the algorithms is as follows:

1. Initialize K clusters randomly.
2. Assign each point in the training data set to the closest cluster.
3. Compute the average of the coordinates of every point associated with each class.
4. Re-locate clusters based on the average of the closest data points.
5. Repeat this process until a certain number of iterations is reached, or until the clusters stop moving.
6. Finally, replace each data point with the coordinates of the closet cluster.

Figure 1 shows the results of the K-means algorithm with $K = 256$. Clearly, the differences between the original picture and the output of the algorithms are hard to notice; they look almost identical. I have tried to let the algorithm run until the clusters converge to a definite position, but it took too long, so I decided to limit the iterations to 10.

(a) Original (b) K-Means

Figure 1: Comparison between the original picture and the compressed picture with k=256.

Additionally, I have run K-means with $K = 4, 8, 16, 32, 64$, and 128. Figure 2 Shows the resulting images. K-means seems to be a simple still effective algorithm for image compression. Even when $K = 32$, the picture looks similar to the original picture. Note, for each distinct K, this experiment was run with 10 iterations. Finally Table 1 quantifies the differences between the generated picture and the original picture. As expected, when the MSE decreases, the PSNR decreases.

Clusters	MSE	PSNR
4	1165.21	17.46
8	370.39	22.44
16	186.08	25.43
32	93.66	28.41
64	62.73	30.15
128	43.63	31.73
256	27.21	33.78

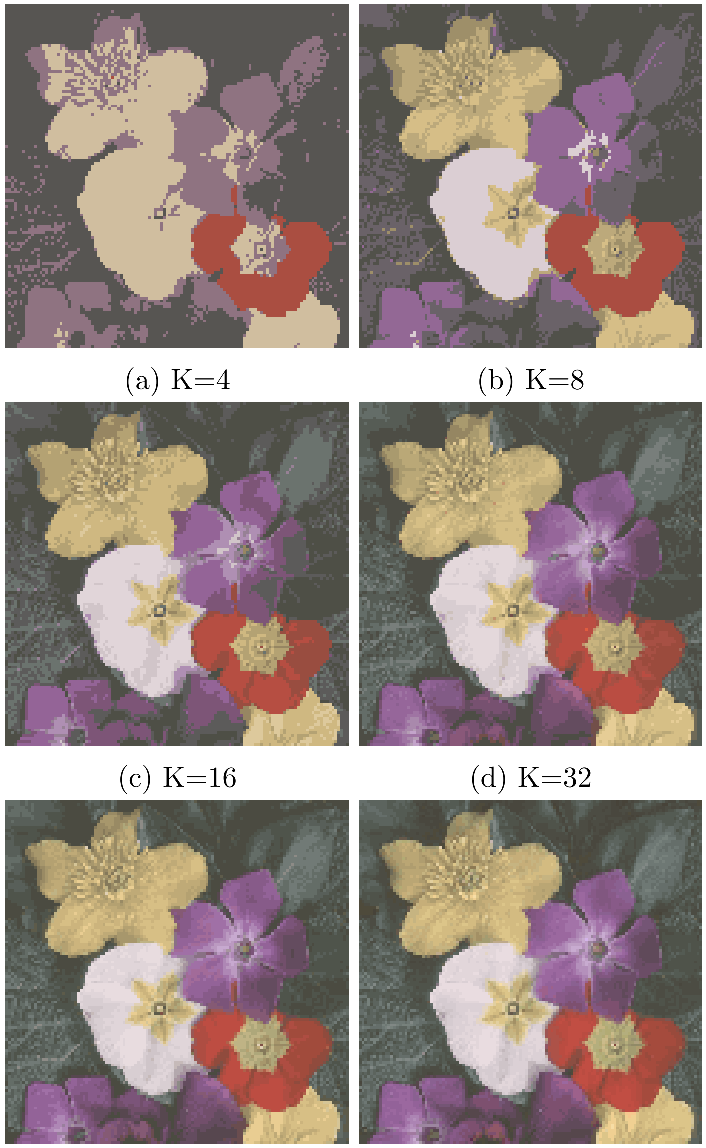
Table 1: K-means Perfomance Table

1.2 Winner-take-all

The second algorithm that I wrote is Winner-takes-all. The steps of my program are as follow:

1. Initialize clusters' coordinates randomly.
2. For each data point in the training dataset, find closest cluster.
3. Update the cluster's coordinate based on the following rule:

$$W_\alpha^{new} = W_\alpha^{old} + \epsilon(X - W_\alpha^{old}) \tag{3}$$



(a) K=4 (b) K=8

(c) K=16 (d) K=32

(e) K=64 (f) K=128

Figure 2: K-means used for image compression with different number of clusters.

4. Repeat this process for a specific number of epochs or until clusters stop changing.

5. Finally, replace each data point with the coordinates of the closet cluster.

After writing the Winner-take-all class, I run the algorithm with $K = 256$. Figure 3 shows the resulting picture in contrast with the original picture. The pictures look similar but in the case of Winner-take-all you can easily see the imperfections. In general, Winner-takes-all performed significantly worst than K-means. This can be observed in the performance metric table, where the MSE values are bigger, and the PSNR are smaller. Figure 4 shows the output of Winner-takes-all with $K = 4, 8, 16, 32, 64,$ and 128.

1.3 Self-Organizing Maps

Finally, I wrote a class for Self-organizing maps. The steps of my algorithm are as follow:

(a) Original

(b) Winner-tales-all

Figure 3: Comparison between the original picture and the compressed picture with k=256.

Clusters	MSE	PSNR
4	2174.98	14.75
8	1596.14	16.10
16	1440.79	16.30
32	1495.75	16.38
64	974.18	18.24
128	724.26	19.53
256	285.89	23.56

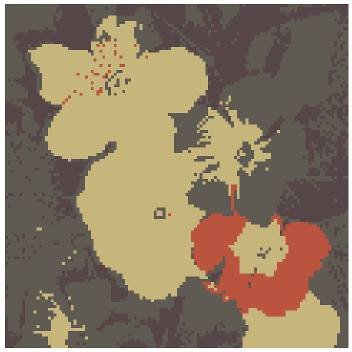
Table 2: Winner-takes-all Performance Table

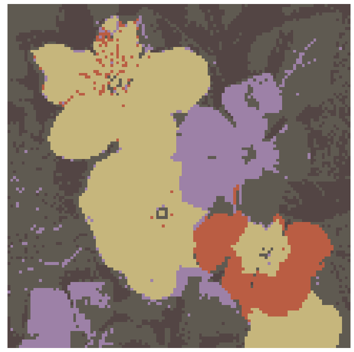
1. Initialize a 2D grid where each neuron represents a cluster.
2. Randomly assign weights to each of the neurons. In other words, assign coordinates in a space of the same dimensions that the dataset.
3. Update neuron's weights using the following rule:

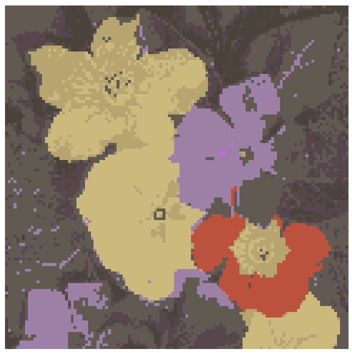
$$W_{\alpha}^{new} = W_{\alpha}^{old} + \epsilon(k)\phi(k)(X - W_{\alpha}^{old}) \tag{4}$$

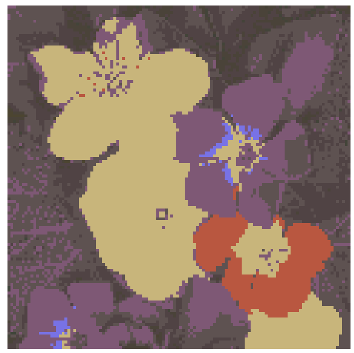
   where $\epsilon(K)$ (the learning rate) can take two forms: $(.9)^t$ or $(1 - \frac{t}{T})$ where t represents current iteration, and T represents the total number of epochs. $\phi(k)$, the neighborhood , is defined as $exp(-\frac{||g_{w_r} - g_{w_{winner}}||^2}{2\sigma^2})$
4. Repeat this process for a specific number of epochs or until clusters stop changing.
5. Finally, replace each data point with the coordinates of the closet cluster.

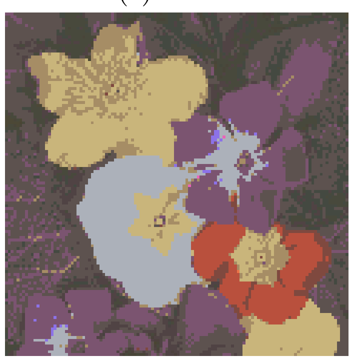
Figure 5 shows the resulting picture after running the algorithm with a grid with 256 neurons. Both pictures look similar but it's easy to tell that the one on the left is the original. Figure 6 displays


(a) K=4


(b) K=8


(c) K=16


(d) K=32


(e) K=64

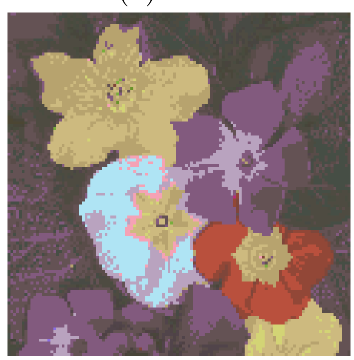

(f) K=128

Figure 4: Winner-takes-all used for image compression with different number of clusters..

the output when the algorithms is run with different numbers of Ks. Consistently with the previous results of this paper, the MSE decreases as the number of clusters increase (see Table 3).

1.4 Mean-shift

I used the sklearn library in order to run the Mean-Shift algorithm. This algorithm automatically detects the optional number of clusters for a given bandwidth. Table 4 shows the experiment results with different bandwidth sizes. Figure 7 provides an additional perspective on the relationship between the bandwidth and the number of clusters.

(a) Original (b) Self-Organizing Maps

Figure 5: Compressed picture with only 256 colors using Self-Organizing Maps.

Clusters	MSE	PSNR
4	6561.12	9.96
8	3670.55	12.48
16	2373.77	14.37
32	1362.11	16.78
64	943.66	18.38
128	549.04	20.73
256	345.92	22.74

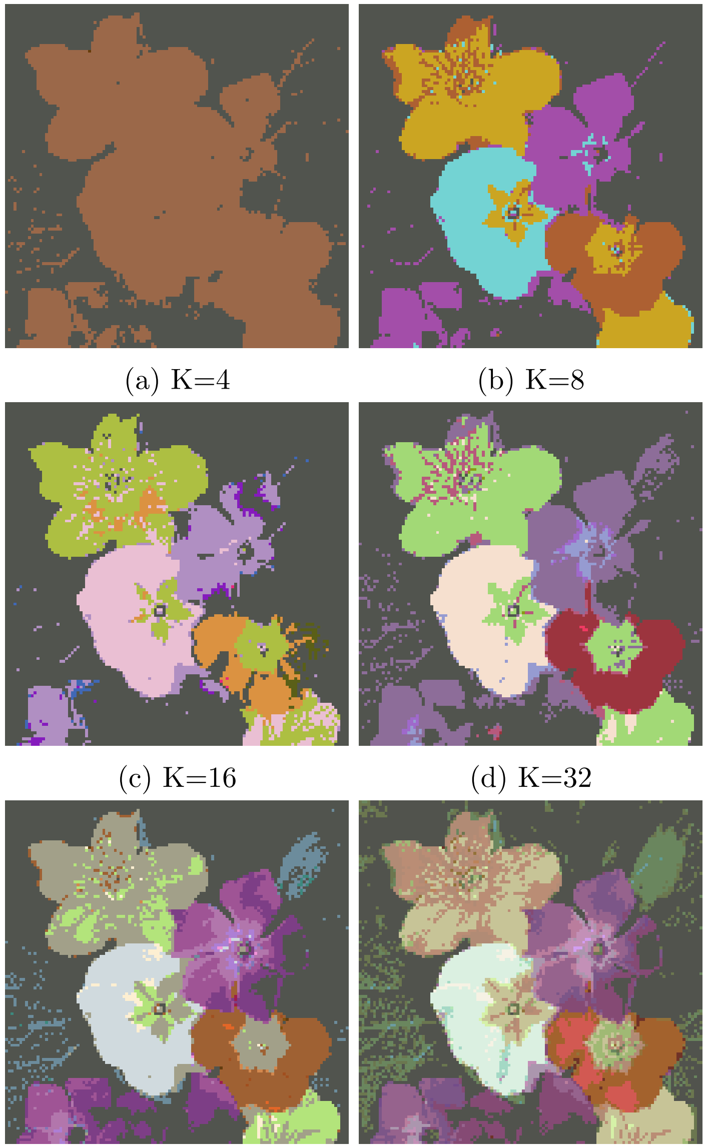
Table 3: SOM Performance Table

Bandwidth	Number of Clusters
2	3435
4	916
6	334
8	146
10	73

Table 4: Bandwidth and Number of clusters provided by Mean-shift.

2 discussion

This paper presents four clustering algorithms for unsupervised tasks. More precisely, the algorithms were applied to compress a picture with 4, 8, 16, 32, 64, 124, and 256 colors. For each of the resulting compressed images, the MSE and the PSNR were reported. When $K = 256$, K-means, Winner-take-all, and SOM reported an MSE of 27.1, 285.89, and 345.92, respectively. Clearly, K-means has performed significantly better than the rest of the algorithms. Additionally, even though computational time results have not been presented, K-means tended to be extremely fast at going



(a) K=4 (b) K=8

(c) K=16 (d) K=32

(e) K=64 (f) K=128

Figure 6: Self-Organizing maps used for image compression with different number of clusters..

through the training dataset. The efficiency of K-means can also be observed by the resulting pictures when different Ks are used (Figure 2). For $K > 8$, the image already looks very similar to the original picture. It's important to mention that the comparison presented in this paper are exclusively in the context of image compression, and the dataset used. Self-Organizing maps reported the worst performance, but at the same time, the algorithm contains a greater parameter space that could be explore extensively. Additionally, different learning rates and neighborhood functions could be implemented. Personally, I think SOM has a huge potential because of the complexity of the algorithm, but this particular application and the hyper-parameters selected do not totally reflect the strengths of the algorithm.

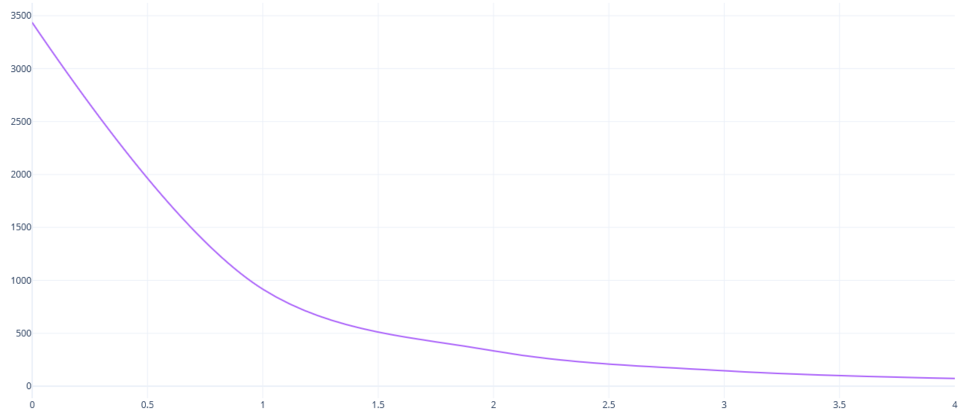


Figure 7: The Y-axes represents the number of clusters, and the X-axes represents the Bandwidth.

References

[1] Amir Ahmad and Lipika Dey *A k-mean clustering algorithm for mixed numeric and categorical data*. pdf.

3 Appendix

3.1 Python Script

```
'''
Created by Kevin De Angeli
Date: 2019-11-14
'''

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sympy as sym
from PIL import Image
import random
import copy
from sklearn.cluster import MeanShift

def readData(showWxample=False):
    #pixels = list(img.getdata())
    #img = Image.open('example2.JPG')
    img = Image.open('pictureData.ppm')
    X = np.array(img)  # 120x120x3 Data
    picShape =[]
```

```python
    picShape.append(X.shape[0])
    picShape.append(X.shape[1])
    X = X.reshape(X.shape[0] * X.shape[1], 3)
    if showWxample == True:
        img.show()
        print("Data Example")
        print(X[0:5])
        print(" ")
        print("Number of entries: ", X.shape[0])
        print("Min of the dataset ", np.amin(X,axis=0))
        print("Max of the dataset ", np.amax(X,axis=0))
        print(" ")
    return X, picShape


def displayPicture(imageArray, picShape):
    newImage = imageArray.reshape((picShape[0],picShape[1],3))
    img = Image.fromarray(newImage, 'RGB')
    img.show()

class K_means(object):

    def __init__(self, data):
        self.X= data
        self.iterations = 0
        self.C = [] #Clusters

    def train(self, k, iterationsLimit = -1):
        '''
        Note 1: the loop will continue until the clusters stop
        changing or until the optional iterationsLimit parameter
        is reached. -1 means no limit.

        Note 2: A common issue is to have empty clusters.
        For this the function clustersUpdate calls reInitializeEmptyClusters


        Steps:
        1. Initialize k clusters at a random position
        2. Label points based on closest neighbor (function: closestCluster)
        3. Update Clusters (function clustersUpdate)

        '''
        #could put low= 0 , high= 256. But I wanted ti try this
        #self.C = [np.random.randint(low=np.amin(self.X), high= np.amax(self.X),
        ↪  size=self.X.shape[1]) for i in range(k)]
```

```
        self.C = [np.random.randint(low=0, high= 256, size=self.X.shape[1]) for i in range(k)]
        self.C = np.array(self.C)
        #self.C = self.C.reshape((k,self.X.shape[1]))

        C_old = np.array([])


        while not self.finishLoopCheck(oldClusters=C_old, iterationsLim=iterationsLimit):
            print("Iteration: ", self.iterations)
            C_old=copy.deepcopy(self.C)  # To copy C by value not by reference
            #print(self.C)

            dataAssignment = self.closestCluster()
            self.clustersUpdate(dataAssignment)

            self.iterations+=1


    def finishLoopCheck(self, oldClusters, iterationsLim):
        '''
        Stop the program if the clusters' position stop changing or
        the limit number of iterations has been reached.
        '''
        if iterationsLim == self.iterations:
            return True
        else:
            return np.array_equal(oldClusters, self.C) #Clusters didn't change ?


    def closestCluster(self):
        '''
        Create a list where each data point is associated with a
        clusters. Then it returns the list of clusters.


        '''
        clusterAssignment = []
        for i in self.X:     #For each dataPoint
            dist = []
            for k in self.C: #For each cluster.
                dist.append(np.linalg.norm(i-k))
            min = np.amin(dist)
            index = dist.index(min)
            clusterAssignment.append(index)
```

```python

        #return a list of size X where each element specifies the cluster.
        return  np.array(clusterAssignment)

    def reInitializeEmptyClusters(self, CIndex):
        '''
        Re-initialize clusters at randon.
        This is used when clusters are empty.
        '''

        newCoordinates = np.random.randint(low=0, high=256, size=self.X.shape[1])
        self.C[CIndex] = np.array(newCoordinates)


    def clustersUpdate(self, clusterAssignments):
        '''
        In order to handle "empty clusters" I re-initialized those clusters randonly.
        '''
        #clusterAssignments = np.array(clusterAssignments)
        newClusterCoordinate=[]
        #update self.C based on clusterAssignments

        for i in range(self.C.shape[0]):
            if i not in clusterAssignments:
                print("Empty Cluster: ", i)
                self.reInitializeEmptyClusters(CIndex= i)
                continue
            findDataPoints = clusterAssignments == i

            dataPointsCoordinates = self.X[findDataPoints]
            newClusterCoordinate = np.average(dataPointsCoordinates,axis=0)
            self.C[i] = newClusterCoordinate

    def mergeDataPoints(self):
        '''
        This function change the value of the
        data points based on the value of the closest neighboor.
        '''
        dataAssignment = self.closestCluster()

        for i in range(self.C.shape[0]):
            selectPoints = dataAssignment == i
            self.X[selectPoints] = self.C[i]

        return self.X
```

```python
class WinnerTakeAll(object):
    def __init__(self, data):
        self.X = data
        self.C = 0
        self.iterations = 0
        self.epselon = 0

    def finishLoopCheck(self, oldClusters, iterationsLim):
        '''
        Stop the program if the clusters' position stop changing or
        the limit number of iterations has been reached.
        '''
        if iterationsLim == self.iterations:
            return True
        else:
            return np.array_equal(oldClusters, self.C) #Clusters didn't change ?

    def closestCluster(self, testPoint):
        '''
        Compute euclidian distance of a testPoit to each of the clusters.
        Return the index of the closest cluster.
        '''
        dist = []
        for i in range(self.C.shape[0]):
            dist.append(np.linalg.norm(self.C[i] - testPoint))
            min = np.amin(dist)
            WinnerIndex = dist.index(min)

        return  WinnerIndex



    def clustersUpdate(self):
        '''
        For each data point. Find the closet cluster (function closestCluster)
        and update the position of the cluster based on W_new = W_old + epselon(X - W_old)
        '''
        newClusterCoordinate=[]
        #update self.C based on clusterAssignments

        for k in self.X:
            winnerCluster = self.closestCluster(k)
            newClusterCoordinate = self.C[winnerCluster] + self.epselon*(k - self.C[winnerCluster])
```

```python
            self.C[winnerCluster] = newClusterCoordinate


    def train(self, k, iterrationsLimit = -1, epselon=0.1):
        '''
        Randomly initialize clusters and then update them.
        '''
        self.epselon = epselon
        self.C = k
        self.C = [np.random.randint(low=0, high=256, size=self.X.shape[1]) for i in
         ↪  range(k)]
        self.C = np.array(self.C)
        C_old = [] #old clusters. Used to check if they stopped changing.

        while not self.finishLoopCheck(oldClusters=C_old,
         ↪  iterationsLim=iterrationsLimit):
            print("Iteration: ", self.iterations)
            C_old = copy.deepcopy(self.C)  # To copy C by value not by reference
            #dataAssignment = self.closestCluster()
            self.clustersUpdate()
            self.iterations += 1


    def closestClusterForMerging(self):
        '''
        Create a list where each data point is associated with a
        cluster. Then it returns the list of clusters.
        This is used to create the final picture.
        Once the cluster position are set, assign the coordinate of
        the cluster to each of the the data points that are close.
        '''
        clusterAssignment = []
        for i in self.X:    #For each dataPoint
            dist = []
            for k in self.C: #For each cluster.
                dist.append(np.linalg.norm(i-k))
            min = np.amin(dist)
            index = dist.index(min)
            clusterAssignment.append(index)

        #return a list of size X where each element specifies the cluster.
        return  np.array(clusterAssignment)

    def mergeDataPoints(self):
        '''
        This function change the value of the
```

```
        data points based on the value of the closest neighboor.
        '''
        dataAssignment = self.closestClusterForMerging()

        for i in range(self.C.shape[0]):
            selectPoints = dataAssignment == i
            self.X[selectPoints] = self.C[i]

        return self.X

class KohonenMaps(object):

    def __init__(self, data):
        self.X = data
        self.xmax= None
        self.ymax= None
        self.learningRate = None
        self.currentEpoch = 1
        self.learnRateFunc = self.timeInverse
        self.totalEpochs = None
        self.C = None #It contains a dic where keys are 2D-grid coordinates, and values are points in the data space.


    def train(self,xmax=10, ymax=10, epochs=4,  learningRate = "time inverse"):
        self.xmax= xmax
        self.ymax= ymax
        self.totalEpochs = epochs
        self.learningRate = learningRate
        if learningRate == "time proportional":
            self.learnRateFunc = self.timeProportional

        self.initializeGrid()

        for k in range(self.totalEpochs): #for each epoch
            for i in self.X: #For each data point
                clusters = np.array(list(self.C.values()))
                gridCoordinates = np.array(list(self.C.keys()))

                winnerIndex = self.findWinnerNeuron(i, clusters)

                self.updateClusters(gridCoordinates[winnerIndex],clusters[winnerIndex],i
                self.currentEpoch+=1


    def updateClusters(self, winnerCoordinates, clusterCoordiante, testPoint):
```

```
        '''
        Based on the winner, update all the clusters.
        :param winnerCoordinates:
        :param clusterCoordiante:
        :param testPoint:
        '''

        winnerCoordinates = np.array(winnerCoordinates)
        coordinateDifference = testPoint - clusterCoordiante[0]

        for i in range(self.xmax):
            for j in range(self.ymax):
                clusterGridCoordinate = np.array([i,j])
                gridDistance = np.sum(np.abs(winnerCoordinates-clusterGridCoordinate))
                newCoordinates = self.updateCordinates((i,j),gridDistance,coordinateDifference)
                self.C[(i,j)] = newCoordinates

    def updateCordinates(self, coordinate, gridDistance,coordinateDifference):
        '''
        Update the cluster based on the eqution:
        W_k+1 = W_k + LearRateFunc()*Neighborhood Function
        Here, the neighbordhood function used is 1/exp(gridDifference/2)
        Note that all clusters are being updated, but the farthest away in the
        2D grid are not being affected much. Some paper use the "radio" idea
        to identify which ones should be updated.

        :param coordinate: 2D grid coordinate
        :param gridDistance: 2D grid distance
        :param coordinateDifference: Difference between the winner cluster and the test point
        :return:
        '''
        #add diferenece as a parameter.
        value = self.C[coordinate]
        newVal = value + (self.learnRateFunc()*(1/np.exp(gridDistance/2))*coordinateDifference)
        return newVal




    def findWinnerNeuron(self, testPoint,clusters):
        '''
        Compute euclidian distance of a testPoit to each of the clusters.
```

```
        Return the index of the closest cluster.
        '''
        dist = []
        for i in range(clusters.shape[0]):
            dist.append(np.linalg.norm(clusters[i] - testPoint))
        min = np.amin(dist)
        #print(dist)
        WinnerIndex = dist.index(min)
        return WinnerIndex #coordinates of the winner


    def initializeGrid(self):
        '''
        Create a grid dictionary where the key is the value in a
        2D matric (i,j), and the key is the coordinates of that point.
        '''
        totalNeurons = self.xmax*self.ymax
        #initialClusters = np.array([np.random.randint(low=0, high= 256,
        ↪  size=self.X.shape[1]) for i in range(totalNeurons)])

        #Create mapping dictionary from grid to coordinates:
        self.C = {}
        for i in range(self.xmax):
            for j in range(self.ymax):
                key = (i,j)
                value = np.array([np.random.randint(low=0, high= 256,
                ↪  size=self.X.shape[1])])
                self.C[key] = value




    def mergeDataPoints(self):
        '''
        This function change the value of the
        data points based on the value of the closest neighboor.
        '''
        clusters = np.array(list(self.C.values()))

        dataAssignment = self.closestCluster(clusters)

        for i in range(clusters.shape[0]):
            selectPoints = dataAssignment == i
            self.X[selectPoints] = clusters[i]
        return self.X

```

```python
    def closestCluster(self, clusters):
        '''
        This function is called by mergeDataPoints only. (for this class)
        Create a list where each data point is associated with a
        clusters (closest). Then it returns the list of clusters.
        '''
        clusterAssignment = []
        for i in self.X:  # For each dataPoint
            dist = []
            for k in clusters:  # For each cluster.
                dist.append(np.linalg.norm(i - k))
            min = np.amin(dist)
            index = dist.index(min)
            clusterAssignment.append(index)

        # return a list of size X where each element specifies the cluster.
        return np.array(clusterAssignment)




    def timeInverse(self):
        '''
        One of the Learning rate functions
        :return: (.9)**k
        '''
        return (.9) ** (self.currentEpoch)

    def timeProportional(self):
        '''
        One of the Learning rate functions
        :return: (1-k/K)
        '''
        return (1 - (self.currentEpoch / self.totalEpochs))



def MSE(A,B):
    '''
    :param A: array of pixels of image 1
    :param B: array of pixels of image 2
    :return: Mean Square Value.
    '''
    mse = np.subtract(A.astype(np.int16), B.astype(np.int16))
    mse = mse**2
    mse = np.sum(mse) / A.shape[0]
```

```
    return mse

def PSNR(A,B):
    '''
    :param A: array of pixels of image 1
    :param B: array of pixels of image 2
    :return: PSNR score.
    '''
    mse = MSE(A,B)
    return 20 * np.log10(255/np.sqrt(mse))



def main():
    data, picShape = readData(showWxample=False)
    originalPic, picShape = readData(showWxample=False)
    print(picShape)




    '''
    kMeans = K_means(data)
    kMeans.train(k=16, iterationsLimit= 10)
    newImage0 = kMeans.mergeDataPoints()
    displayPicture(newImage0,picShape )
    print(MSE(originalPic,newImage0))
    print(PSNR(originalPic,newImage0))
    '''




    '''
    #data, picShape = readData(showWxample=False)
    winner_take_all = WinnerTakeAll(data)
    winner_take_all.train(k=16, iterrationsLimit= 10, epselon = 0.18) #.1 works
    newImage2 = winner_take_all.mergeDataPoints()
    displayPicture(newImage2, picShape)
    print(MSE(originalPic,newImage2))
    print(PSNR(originalPic,newImage2))
    '''



    '''
    SOM = KohonenMaps(data)
    SOM.train(xmax=4, ymax=4, epochs=5, learningRate="time inverse")
```

```
    newImage3 = SOM.mergeDataPoints()
    displayPicture(newImage3,picShape)
    print(MSE(originalPic,newImage3))
    print(PSNR(originalPic,newImage3))
    '''

    bandwidth=10
    ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
    ms.fit(data)
    labels = ms.labels_
    cluster_centers = ms.cluster_centers_
    labels_unique = np.unique(labels)
    n_clusters_ = len(labels_unique)
    print("With a bandwidth of size: ", bandwidth, "Number of clusters: ",
     ↪  n_clusters_)

if __name__ == "__main__":
    main()
```