Machine Learning
CSE 8673

Programming Assignment 2

Student: Anh Do
NetID: aqd14

Part A: Deliverables

Training set	**Number of misclassification**	**Fraction of misclassification**
960 documents	5	0.0192
50 documents	7	0.0269
100 documents	6	0.023
400 documents	6	0.023

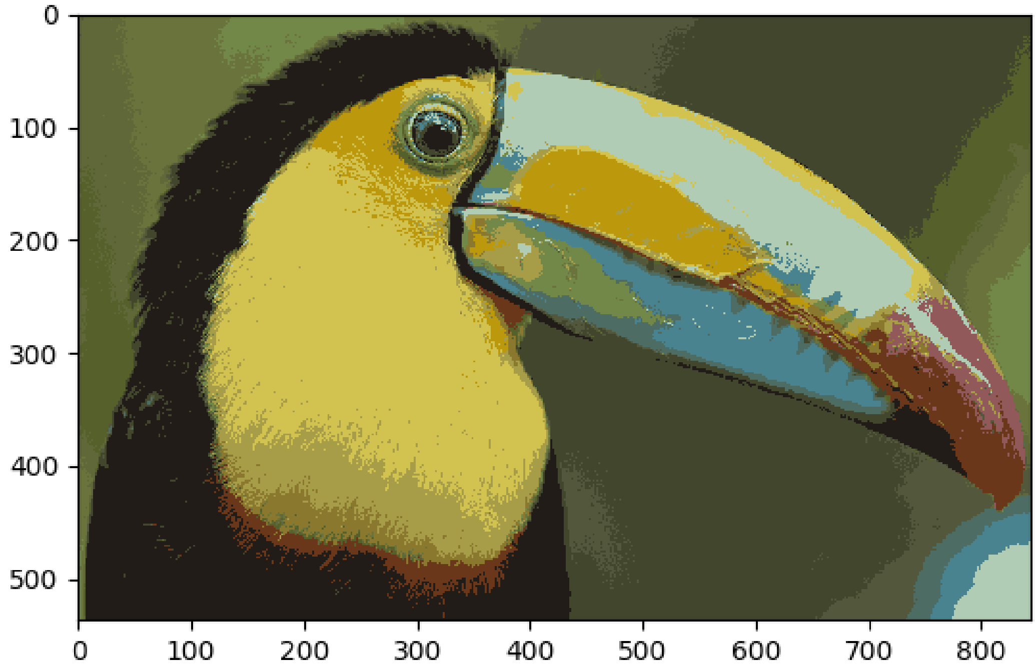
Part B: Deliverables

1. Repeat the entire k-means process for k = 2, . . . 15. Include the modified images after each run. How do the images differ from one another? Does the quality of the reproduction noticeably improve as k increases?

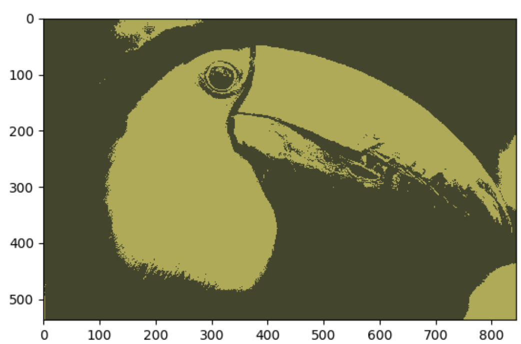
Solution:

The figure becomes more realistic when the number of colors increases. With more color, we can reproduce the image that effectively reflects the original.

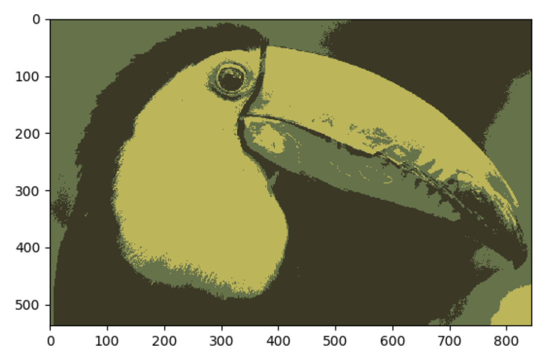
K = 16


0
100
200
300
400
500
0
100
200
300
400
500
600
700
800

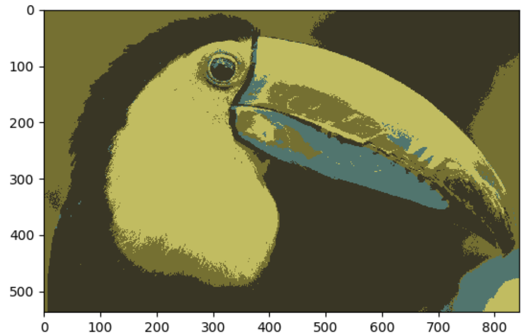
K = 2


0
100
200
300
400
500
0
100
200
300
400
500
600
700
800

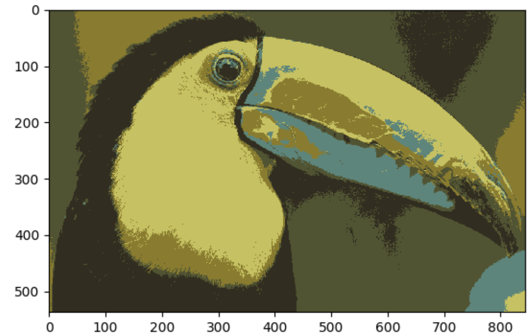
K = 3


0
100
200
300
400
500
0
100
200
300
400
500
600
700
800

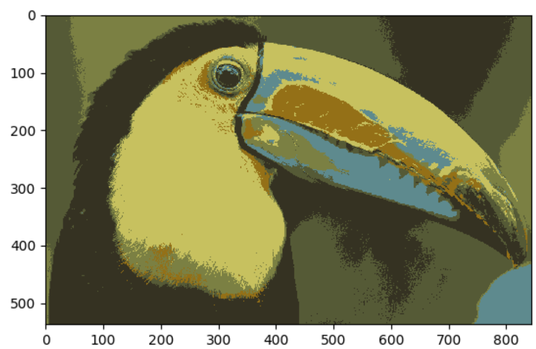
K = 4



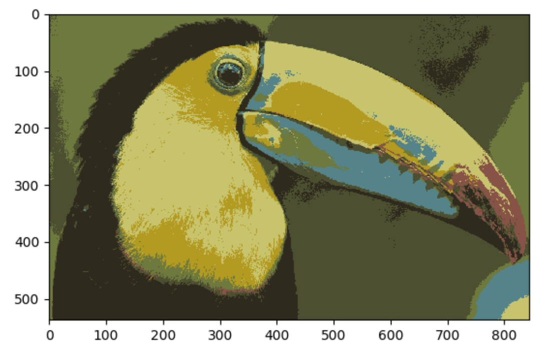
K = 5



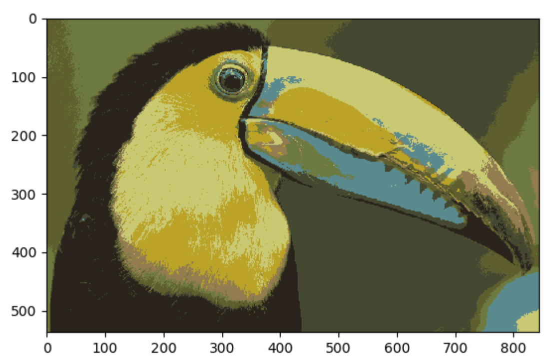
K = 6



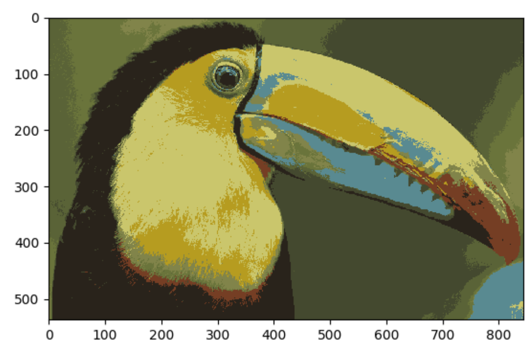
K = 7

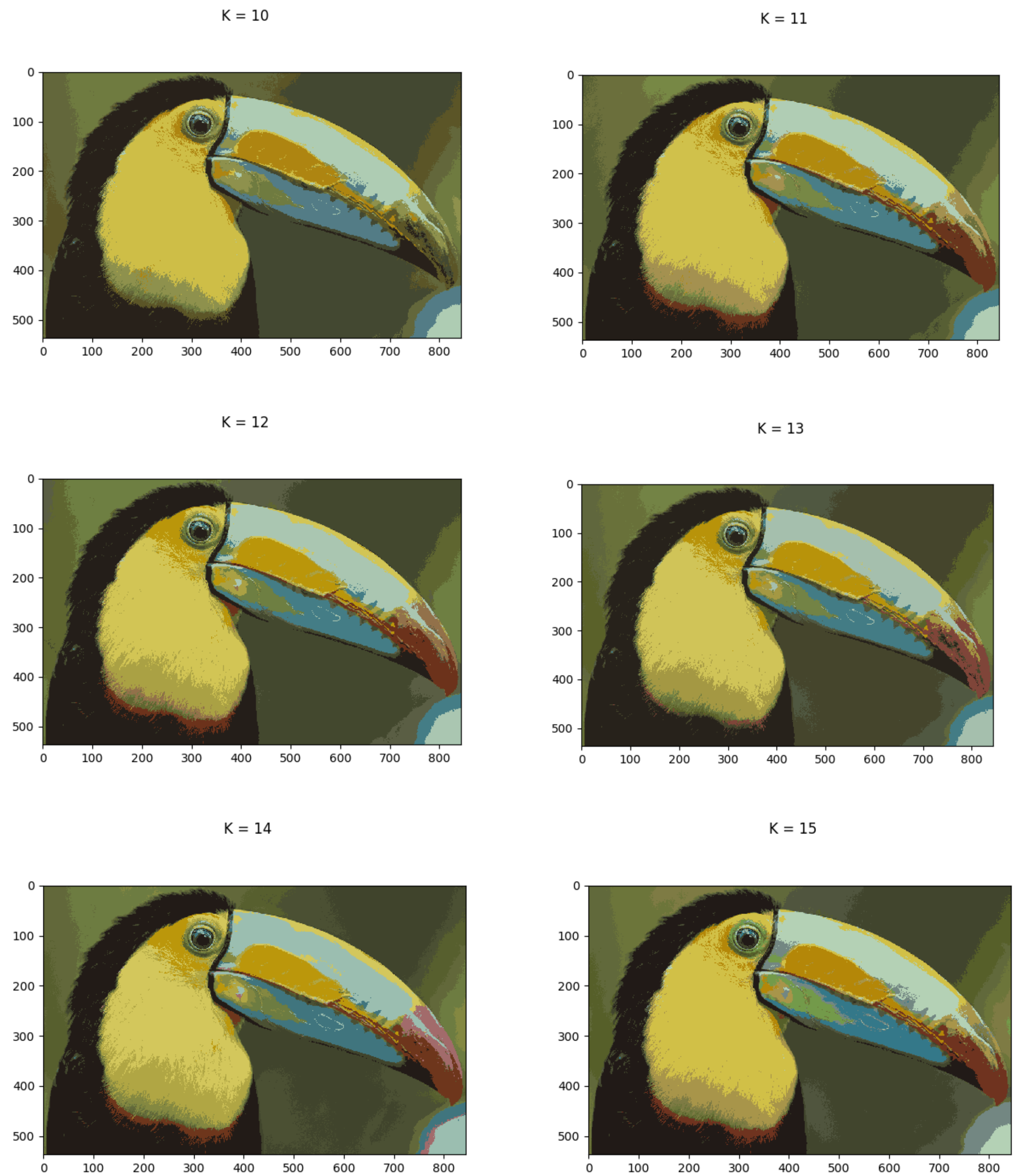


K = 8



K = 9



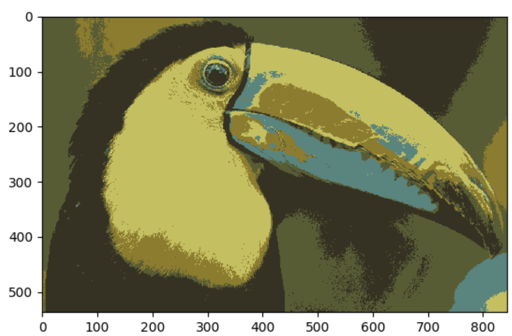


2. Select a single k between 2 and 15 and run the k-means process several times. Include the modified images from these runs. Does the algorithm find the same cluster centroids each time? Are any differences in centroid locations noticeable in the modified images?

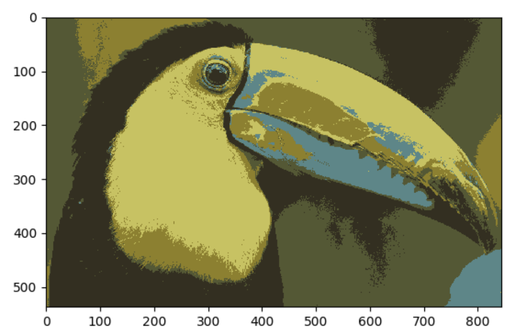
Solution: The k-means algorithm found different sets of centroids for each run. The reason is that the algorithm only considers the mean RGB values and Euclid distance when finding the cluster for each pixel in image. However, the generated images look almost the same.

Attempt	K	Iterations	Centroids
1	5	25	[[140. 124. 47.] [196. 191. 97.] [90. 132. 126.] [88. 92. 53.] [53. 50. 35.]]
2	5	41	[[199. 194. 99.] [84. 88. 53.] [140. 128. 49.] [94. 134. 136.] [51. 47. 33.]]
3	5	28	[[97. 139. 136.] [87. 90. 52.] [140. 128. 50.] [200. 194. 96.] [52. 48. 34.]]
4	5	27	[[87. 130. 126.] [55. 52. 36.] [197. 192. 98.] [139. 128. 49.] [91. 93. 52.]]

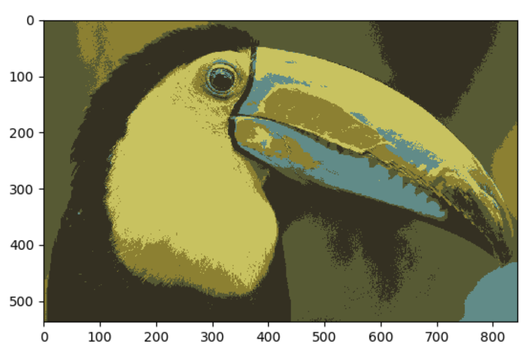
Attempt 1 with k = 5



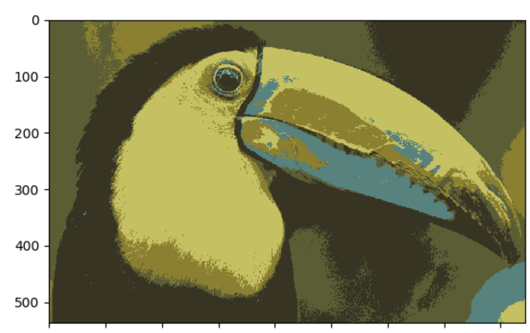
Attempt 2 with k = 5



Attempt 3 with k = 5



Attempt 4 with k = 5



3. If you were to run the code with k = 1 how many iterations would it take to converge? What would the single centroid correspond to?

Solution: It will take only two iterations to converge. The single centroid is the mean of total pixels in the image.

Code Appendix

Naïve_bayes.py

```
'''
Created on Sep 29, 2017
naive_bayes.py
@author: aqd14
'''

from __future__ import division
import numpy as np
from scipy import misc
from scipy import sparse as sps
import matplotlib.pyplot as plt

numTokens = 2500

class MultinomialNaiveBayes():
    def __init__(self):
        self.py_pos = 0.0   # estimates the probability that a particular
word in a spam email will be the k-th word in the dictionary
        self.py_neg = 0.0   # estimates the probability that a particular
word in a non-spam email will be the k-th word in the dictionary
        self.phi_pos = 0.0  # the probability that any particular email will
be a spam email

    def fit(self, train_labels, train_matrix, num_tokens):
        # Training phase
        numTrainDocs = train_labels.shape[0]
        spam_email_pos = np.where(train_labels==1)      # array-like:     The
indices of spam emails
        nonspam_email_pos = np.where(train_labels==0)    # array-like:     The
indices of non-spam emails
        email_word_count = np.sum(train_matrix, 1)      # array-like:     The
total word count for each email
```

```
        # Calculate phi_k|y=1 = p(xj = k|y = 1)
        self.py_pos = (train_matrix[spam_email_pos].sum(axis=0) + 1) /
(np.sum(email_word_count[spam_email_pos]) + num_tokens)
        # Calculate phi_k|y=0 = p(xj = k|y = 0)
        self.py_neg = (train_matrix[nonspam_email_pos].sum(axis=0) + 1) /
(np.sum(email_word_count[nonspam_email_pos]) + num_tokens)

        # prior
        self.phi_pos = np.count_nonzero(train_labels)/numTrainDocs

    def predict(self, test_labels, test_matrix):
        num_test_docs = test_labels.shape[0]
        log_p_pos = test_matrix.dot(np.log(self.py_pos.T)) +
np.log(self.phi_pos)
        log_p_neg = test_matrix.dot(np.log(self.py_neg.T)) + np.log(1 -
self.phi_pos)

        results = log_p_pos > log_p_neg
        # Convert from True/False to 1/0
        return np.squeeze(np.asarray(results.astype(dtype=int)))

def train_and_test(files):
    # Extract parameters
    train_labels_f = files[0]
    train_features_f = files[1]
    test_labels_f = files[2]
    test_features_f = files[3]

    # Load the labels for the training set
    train_labels = np.loadtxt(train_labels_f,dtype=int)
    # Get the number of training examples from the number of labels
    numTrainDocs = train_labels.shape[0]
    # This is how many words we have in our dictionary
    # Load the training set feature information
    M = np.loadtxt(train_features_f,dtype=int)
    # Create matrix of training data
    train_matrix = sps.csr_matrix((M[:,2], (M[:,0], M[:,1])),
shape=(numTrainDocs, numTokens))

    classifier = MultinomialNaiveBayes()
    classifier.fit(train_labels, train_matrix, numTokens)

    test_labels = np.loadtxt(test_labels_f, dtype=int)
    # Load the test set feature information
    N = np.loadtxt(test_features_f,dtype=int)
```

```python
    # Create matrix of test data
    test_matrix = sps.csr_matrix((N[:,2], (N[:,0], N[:,1])))

    prediction = classifier.predict(test_labels, test_matrix)

    num_wrong_docs = np.sum(prediction != test_labels)

    print('Number of wrong classification = {0}'.format(num_wrong_docs))
    print('Fraction of wrong classification =
{0}\n\n'.format(num_wrong_docs/test_labels.shape[0]))


def main():
    files = ['pa3data/train-labels.txt', 'pa3data/train-features.txt',
'pa3data/test-labels.txt','pa3data/test-features.txt']
    print('Working with 960-document dataset...')
    train_and_test(files)

    files = ['pa3data/train-labels-50.txt', 'pa3data/train-features-50.txt',
'pa3data/test-labels.txt','pa3data/test-features.txt']
    print('Working with 50-document dataset...')
    train_and_test(files)

    files = ['pa3data/train-labels-100.txt', 'pa3data/train-features-
100.txt', 'pa3data/test-labels.txt','pa3data/test-features.txt']
    print('Working with 100-document dataset...')
    train_and_test(files)

    files = ['pa3data/train-labels-400.txt', 'pa3data/train-features-
400.txt', 'pa3data/test-labels.txt','pa3data/test-features.txt']
    print('Working with 400-document dataset...')
    train_and_test(files)

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

kmeans.py

```python
'''
Created on Oct 1, 2017

@author: aqd14
'''

import numpy as np
```

```python
import matplotlib.pyplot as plt
from scipy import misc

def init_centroids(A, n_clusters):
    """
    Initialize centroids for pixels in RGB mode (ranging from 0 to 255).
    Randomly pick n_clusters points in the original image to be centroids.

    Parameters
    ----------
    n_clusters : int
        Number of expected clusters
    A : 3-d matrix
        The pixels in image and their coordinates
    Returns
    -------
    centroids : array-like
        A randomly initialized centroids ranging from 0 to 255
    """
    centroids = A[np.random.choice(A.shape[0], n_clusters, replace=False),
                  np.random.choice(A.shape[1], n_clusters, replace=False), :]

    return centroids

def init_cluster(centroids):
    """Initialize cluters

    Parameters
    ----------
    centroids : 2-d array
        List of centroids

    Returns
    -------
    clusters : dictionary
        Mapping from centroids to a list of points in clusters
    """
    clusters = {}

    for c in range(centroids.shape[0]):
        clusters[c] = []
    return clusters

def assign_cluster(A, clusters, centroids):
    """Assign nearest cluster for all pixels
```

```
    Parameters
    ----------
    A : RBG matrix representation for image

    clusters : dictionary
        List of centroids associated with their points in clusters

    centroids : 2-d array
        Current centroids
    """
    # Euclid distance from given point to the centroids
    for i in range(A.shape[0]):
        for j in range(A.shape[1]):
#             assign_cluster(clusters, centroids, A[i][j])
            pixel = A[i][j]
            distance = np.sum((centroids - pixel) ** 2, axis=1)
            # Assign closest cluster for the given pixel
            min_index = np.argmin(distance)
            clusters[min_index].append(pixel)

def update_centroids(clusters, centroids):
    new_centroids = np.zeros((centroids.shape[0], centroids.shape[1]))
    for c in range(centroids.shape[0]):
        points = np.asarray(clusters[c])
        if len(points) > 0:
            new_centroids[c] = np.round(np.mean(points, axis=0))
        else:
            # a centroid without any points
            new_centroids[c] = centroids[c]
    return new_centroids

def kmeans(A, n_clusters, max_iter=100, tolerance=1e-5):
    """Simple implementation for K-Means algorithm to compress an image by
reducing the number of colors it contains

    Parameters
    ----------
    A : RGB matrix representation for image

    n_clusters : int
        Number of color clusters

    max_iter : int
        Maximum number of iteration for finding centroids
```

```
        Default value is 100

    tolerance : float
        The minimum Euclid distance of centroids values between two
consecutive iteration to be considered converged

    Returns
    -------
    centroids : 2-d array
        Converged centroids
    """
    centroids = init_centroids(A, n_clusters)  # default centroids
    clusters = init_cluster(centroids)  # np.zeros((A.shape[0], A.shape[1],
1))    # store the index of centroids for each pixel
    ite = 1
    while(ite <= max_iter):
        # print('Iteration {0}'.format(ite))
        assign_cluster(A, clusters, centroids)
        update_centroids(clusters, centroids)
        new_centroids = update_centroids(clusters, centroids)

        err = np.sqrt(np.sum((new_centroids - centroids) ** 2))
        # print('Error = {0}\n'.format(err))
        if err < tolerance:
            print('Converged after {0} iterations!'.format(ite))
            break;
        centroids = new_centroids
        ite += 1

    return centroids

def compress_image(B, centroids):
    """Replace each pixel in the image with its nearest cluster centroid
color

    Parameters
    ----------
    centroids : 2-d array
        Convergered centroids

    B : RBG matrix image
        Image to be compressed

    Returns
    -------
```

```
    B : RBG matrix image
        Compressed image
    """
    for i in range(B.shape[0]):
        for j in range(B.shape[1]):
            pixel = B[i][j]
            distance = np.sum((centroids - pixel) ** 2, axis=1)
            # Assign closest cluster for the given pixel
            min_index = np.argmin(distance)
            B[i][j] = centroids[min_index]

def main():
    A = misc.imread('pa3data/b_small.tiff', mode='RGB')

    for n_clusters in range(1, 17):
        centroids = kmeans(A, n_clusters)
        print('Centroid for clusters {0} are {1}'.format(n_clusters,
centroids))
        B = misc.imread('pa3data/b.tif', mode='RGB')
        compress_image(B, centroids)
        plt.imshow(B)
        plt.suptitle('K = {0}'.format(n_clusters))
        plt.savefig('figures/kmeans' + str(n_clusters) + '.png')


    n_clusters = 5
    for i in range(4):
        centroids = kmeans(A, n_clusters)
        print('Centroid for clusters {0} are {1}'.format(n_clusters,
centroids))
        B = misc.imread('pa3data/b.tif', mode='RGB')
        compress_image(B, centroids)
        plt.imshow(B)
        plt.suptitle('Attempt {0} with k = {1}'.format(i+1, n_clusters))
        plt.savefig('figures/kmeans_' + str(n_clusters) + '_' + str(i+1) +
'.png')

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