Group III

Image Analysis Group **Report 2**

CS3IA16: Image Analysis

University of Reading

Created by:

Shavin Croos: 27015244

Dan Keep: 28016852

Uthman Maigari: 28017661

Date of Completion: 10/01/2022

Actual hrs spent: 9+ hours

Assignment Evaluation:

Abstract

The objective of this coursework is to compress, then decompress 3 given images using one of 2 compression types, being either lossy or lossless. Lossy compression is one way of compressing information such that the file size is decreased as a result of the removal of information in the file. This reduces the image quality and is also an irreversible process. On the other hand, lossless compression is another way of compressing information, but the original information can be nicely restored out of the compressed information. This courseworks checks the fundamentals of image compression and aims to return basic information regarding the features, application and tools of the compression carried out.

Introduction

Image compression is a method of data compression which is commonly applied to computerised images in order to save storage space [1]. The method has been around since the 1940s, beginning with entropy coding [1]. The 1960s saw the introduction of transform coding, but it wasn't until the 1970s that image data compression saw key development with the launch of discrete cosine transform (DCT), a lossy compression method, which has become the basis for image file format JPEG [1]. Thanks to this method, the production of JPEG images increased to the point where 7 billion of them are created per day as of 2015 [1]. Wavelet coding was developed as a result of DCT, which worked the same as DCT, but using wavelets rather than DCT's block based algorithm [1].

The main objective is to develop algorithms of choice to compress, then decompress the 3 given images to us such that they occupy the least amount of storage, but preserving the quality. Like with the first coursework, the mean square error (MSE) will evaluate the result and it is used to compare image compression quality. It represents the cumulative squared error between compressed and original images. However, another value called the peak signal-to-noise ratio (PSNR) value will also be calculated, which uses the MSE value obtained in the calculation. This is to further ensure that the compression-and-decompression cycle had worked correctly and is returning the expected outcome. The algorithms developed for image compression that were chosen for this coursework were the Principal Component Analysis (PCA), Singular Value Decomposition (SVD) and K-Means Clustering.

Methodology

Thomas Methodology

Singular Value Decomposition Theory

For my image compression, I decided to use the Singular Value Decomposition (SVD) method. In theory, the SVD method works by dividing a given matrix into three key smaller matrices to display the information. Take for example the matrix B of size $r \times c$, where r denotes the number of rows in the matrix and c denotes the number of columns in the matrix. B can be divided into three smaller matrices in the form of the equation $B = UDV^T$, where the size of U is $r \times r$, the size of D is $r \times c$ and D is diagonal and the size of V^T is $c \times c$. Now for the matrix multiplication to occur, we must ensure that the column size of the first matrix is the same as that of the row size

of the second matrix. The output matrix will be $x \times z$ when a matrix of size $x \times y$ is multiplied with a matrix of size $y \times z$. If the sub-matrices are used from the matrix B, the multiplication will return a matrix with the same size:

$$r \times c = [(r \times r)(r \times c)](c \times c) = (r \times c)(c \times c) = (r \times c)$$

The key aspect of the matrices UDV^T is that the information is organised as such that the key information is placed at the top. In this case, U holds the key information regarding the rows of the matrix and the key information regarding the matrix is contained in the first column. V^T holds the key information regarding the columns of the matrix and the key information regarding the matrix is contained in the first row. D is a diagonal matrix which only contains at most "r" key values, with all the other values of the matrix being zero. Since this matrix's key numbers are only stored on the diagonal, it is safe to ignore this fact for the size comparison.[2]

Continuing from the size case, if the key information of U is stored on its first column, then U's key information can be written as an $(r \times 1)$ matrix. If the key information of V^T is stored on its first row, then V^T's key information can be written as a $(1 \times c)$ matrix. We will also say that the key information of D is stored on the first row, first column of that matrix, yielding a (1×1) matrix. If the new matrices are now multiplied:

$$U' D' V^{T'} = [(r \times 1)(1 \times 1)](1 \times c) = (r \times 1)(1 \times c) = (r \times c)$$

The overall output size is the same as that of the original matrix. The final matrix obtained, which will be called B', is a good estimate of the original matrix B. [2]

Singular Value Decomposition Implementation

By utilising the theory mentioned earlier, the SVD method can also be applied to an image. The implementation begins with loading a colour image from the chosen directory, along with initialising the colour variables. In the accessImage function, the colour image is opened and is divided into 3 matrices which correspond accordingly to the 3 main colour channels of the image, which are red, green and blue (RGB channels). The RGB matrices are returned and are displayed to show what the colour channels look like in picture form prior to compression.

After this, the singular values limit (SVL) is initialised and set to determine the singular values to keep during the compression and use when reconstructing the compressed image. In the compress method, each of the RGB matrices are subdivided into the new submatrices U, D and V and these submatrices undergo SVD compression through the numpy.linalg.svd() function. Most of the values of the submatrices post-SVD are set to 0 and the k variable uses the SVL value to determine the range of values to keep.

Once all this has been done, the new compressed matrices are then multiplied with each other , using the numpy.matmul function (with the SVL limit being applied to each matrix), in order to reconstruct the compressed image. Setting the compressed image to 'uint8' will allow for the compressed image to regain its colour for each block. The compressed matrices are then exported in order to create the image memory for each of the colour channels needed for the

reconstruction. Finally, the image memory of the compressed matrices are finally merged together to create the new compressed image.

After the SVD compression has taken place, the original image and the compressed image's dimensions are called and the size of the 2 images are calculated. These values are then used to calculate the compression ratio, which is determined by dividing the compressed image size by the original image size (the value is also multiplied by 100 to get the ratio as a percentage). The compressed image is then saved to the same directory that the original image was called from.

In addition to calculating the compression ratio of the image, further statistical calculations are carried out on the image. The first one is the mean square error (MSE), which specifically measures the cumulative squared error between the compressed and the original image. The lower the number of this statistic, the lower the error between the images. This is the first measure to check the image compression quality after the compression has taken place. The second method is what is known as the peak signal-to-noise ratio (PSNR), which measures the peak error. The higher this value, the better the quality of the compressed image. This is the second measure to check the image compression quality, which uses the value obtained from the MSE calculation to obtain the result.

Unfortunately, as the SVD is a lossy compression technique, the process cannot be reversed to obtain the perfect image again (e.g. doing the inverse of SVD). This means that there is no actual decompression method for the SVD.

The SVD compression program was constructed using Python and the following libraries:

- PIL (to import the Image module)
- NumPy (incl. np)
- MatPlotLib (to import the Pyplot module)
- Math

Uthman Methodology

By using K-Means, a Machine Learning clustering algorithm, the script clusters all the colours in an image into 16 clusters and replaces the RGB value of every pixel with the RGB value of its corresponding cluster centre, thus reducing the amount of memory that must be set aside for saving each image.

In order to segregate the dataset and place them into respective clusters, we choose k as the number of clusters, so K = 2. The cluster will be formed by choosing some random two points to act as centroids. The distance between each data point and the nearest K-point or centroid will now be used to assign each data point to a scatter plot. This will be accomplished by drawing a median between both centroids. The distance between the data points and the centroids of each datapoint is determined. By calculating this distance, we can determine to which cluster

the points belong. Re-calculating the centroids which is calculating the new centroid values. When the difference between the old and new centroids becomes negligible, stop the algorithm.

The compressed image quality is measured relative To the original. Based on MSE measure, the Peak Signal-to-Noise Ratio (PSNR) is commonly used to judge the quality Of a compressed image relative to the uncompressed image. PSNR, MSE, SSIM performance measures are used in this research To compare the original and compressed image according to Various K values.

The Libraries requirements for the k means algorithm include:

- Scipy
- Numpy
- Math
- Matplotlib
- from sklearn.cluster import KMeans

Dan Methodology

Kmeans lets us cluster pixels into randomly chosen centroids that average the pixel values to that of the cluster centre and therefore reduce the number of bits to represent the image.

I also used colour reduction which reduces the amount of dimensions to each colour channel which are merged back together to reduce the number of bits.

Results and Discussion

Thomas Results and Discussion

In order to find the right SVL value for compression to be as smooth as possible, the SVL value was increased in increments of 20 and tested against the original image to see how close the compressed image was. The testing began with the SVL value of 20.

Figure 1: Image 3 size 3024 × 4032 with SVL values {{20, 40, 60, 80}, {100, 120, 140, 160}}

Through trial and error, as the SVL value increased, the clearer the final compressed image looked each time. By the time the SVL limit was at 160, the compressed image looked a lot sharper, which suggests that the contrast returned may be stronger than with the original image. The trees and the sky in the SVL levels of 140 and 160 look more refined upon close inspection, but if looked at a glance, the compressed images with SVL values from 80 to 160 look very similar to each other. Since the contrast looks sharper everytime the SVL value increases, it can be said that the SVD is more pattern focused than other compression methods.

After this, it was decided that the SVL value for the final compression of the image would be kept at 160, as once the precision point was hit, the difference between the original image and the compressed image with SVL of 160 was pretty much indistinguishable. This is due to the fact that the error rate between these two images has become insignificant enough that this isn't an issue for the large image anymore.

Figure 2a and 2b: Left image shows the original image 3 and the right image shows the final compressed image 3.

After carrying out the compression of the image, the size of the original image and the compressed image were calculated using height and width of the images obtained separately. Since the images are in colour, the image size for both images had to be multiplied by 3. The formula used for calculating the sizes are as shown:

$$\text{Original Size} = \text{Original Height} \times \text{Original Width} \times 3$$
$$\text{Compressed Size} = \text{SVL} \times (1 + \text{Compressed Height} + \text{Compressed Width}) \times 3$$

After doing the calculations, these are the sizes that were given for image 3:

Original Size = 36578304 bytes (36.58 MB [approx. 2.d.p])
Compressed Size = 3387360 bytes (3.39 MB [approx. 2.d.p])

This coursework also required me to find the compression ratio for image 3, which was obtained by dividing the Compressed Size with the Original Size. When the sizes obtained are plugged into the equation, the compression ratio of image 3 is 0.0926 (approx. 3.s.f). This means that the final compressed image size is 9.26% of the original image size.

Once this was done, some further measures were carried out on the images. The MSE was calculated between the original image and the compressed image and the result obtained was 38.41 (approx. 2.d.p), which suggests that the cumulative square error between the images is very low, meaning that the images are almost quite similar. However, when the MSE is plugged into the PSNR calculation, the result returned -15.84 (approx. 2.d.p), which tells us that the quality of the compressed image returned is quite poor. This goes to show that the SVD technique had caused irreversible data loss to the compressed image and thus cannot be properly reverted back to its former state.

Uthman Results and Discussion

Demonstration of work, Image quality before compression technique was used:

After reconstructing the image, assigning every pixel the rgb colour of their label's centre.

Below is the calculation of the compression ratio,

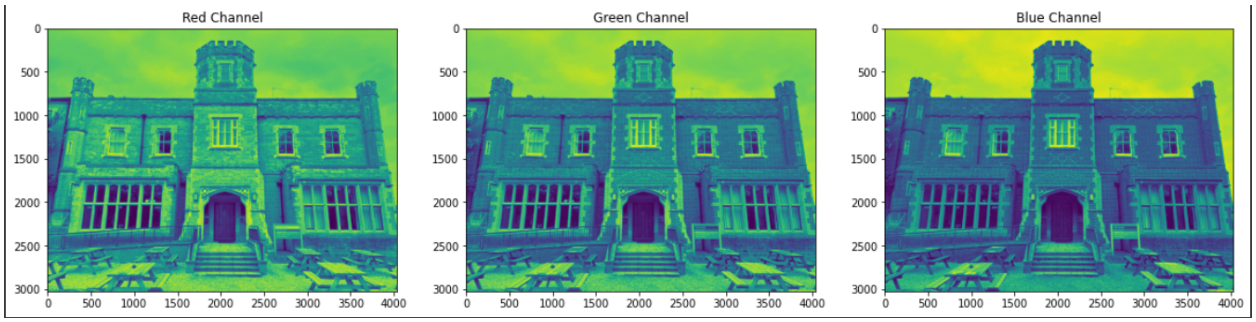
```
# CALCULATION AND DISPLAY OF COMPRESSION RATIO

mr = h
mc = w
singularValuesLimit = 160

# originalSize = mr * mc * 3
# compressedSize = singularValuesLimit * (1 + mr + mc) * 3

print('original size:')
print(originalSize)
print('\ncompressed size:')
print(compressedSize)
print('\nRatio compressed size / original size:')

ratio = compressedSize * 1.0 / originalSize

print(ratio)
print('\nCompressed image size is ' + str( round(ratio * 100 ,2)) + '% of the original image ' )
print('DONE - Compressed the image! ')
```

```
original size:
36578304

compressed size:
3387360

Ratio compressed size / original size:
0.09260571512555639

Compressed image size is 9.26% of the original image
DONE - Compressed the image!
```

The K-Mean clustering technique is presented in this report in general, then how to use it in image compression. The results demonstrate the benefits from compressing the images by reducing the size of the images while preserving the acceptable quality of the compressed images. For future work we intend to extend the images variety to include, measuring the quality of the proposed algorithm on high resolution images as well as, measuring the quality of night mode images. In addition we intend to expand the images format to include HEIF and HEVC. The codes are provided in the appendix.

Dan Results and Discussion

Dimensional reduction



Separate colour channels.

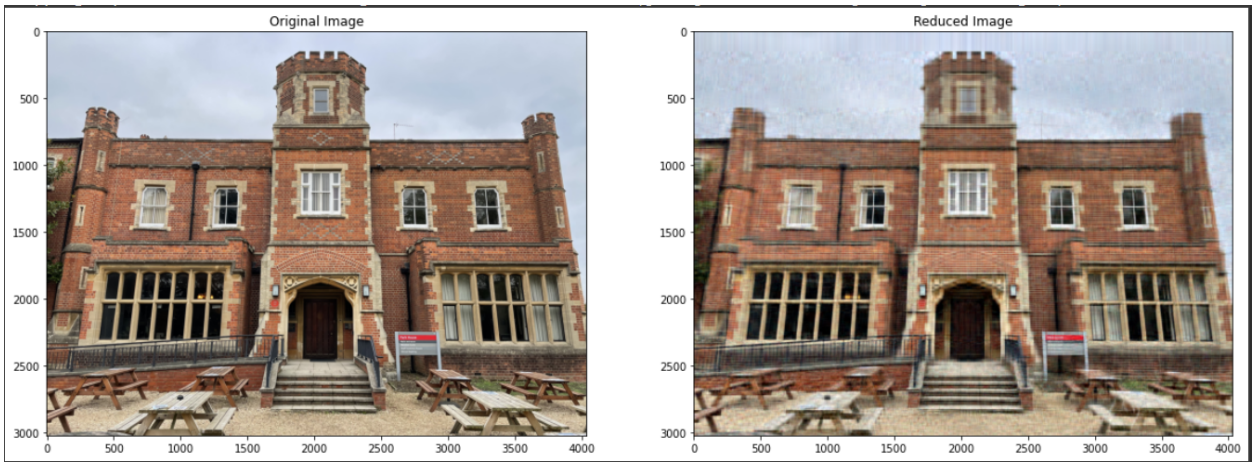
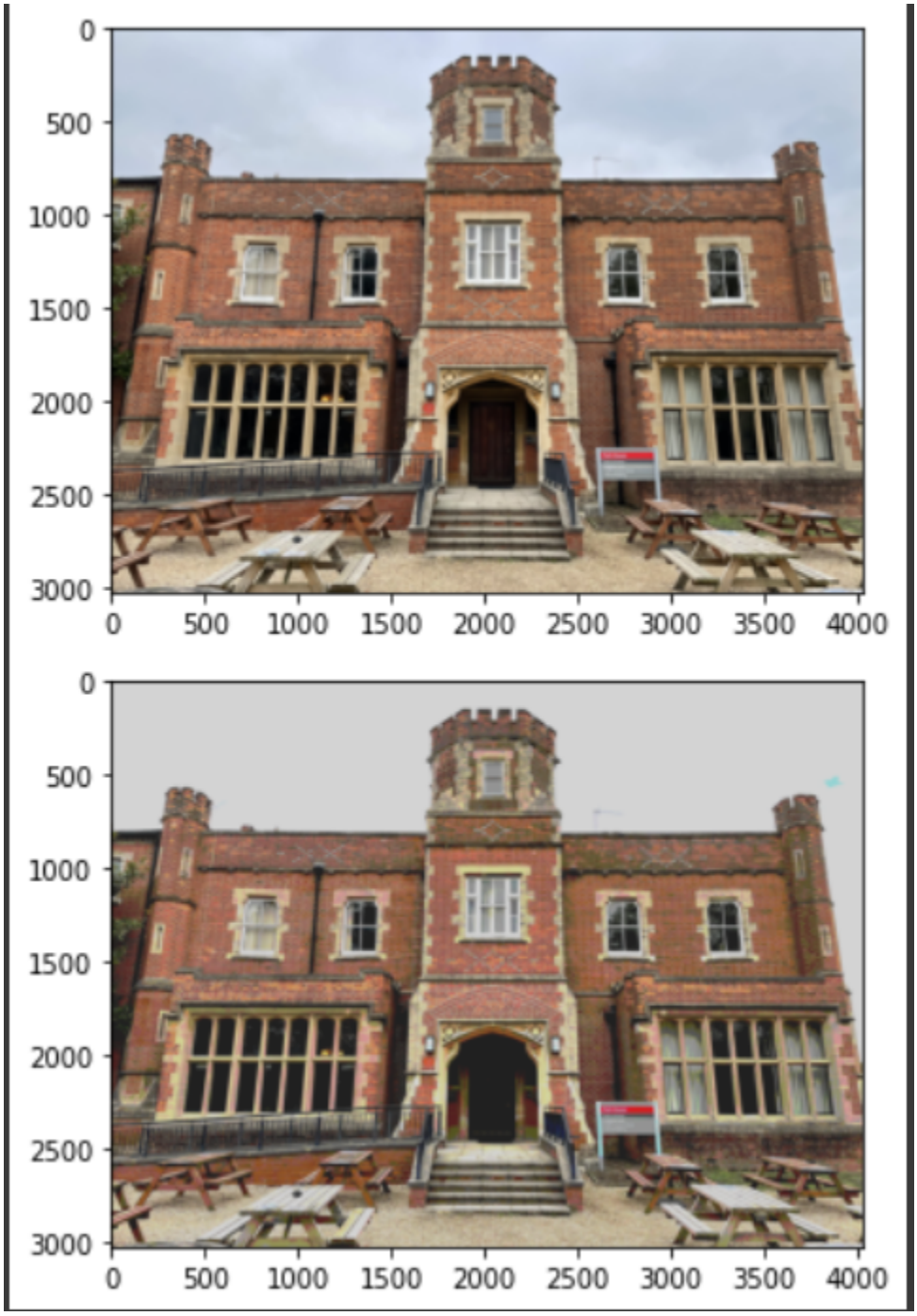


Image comparison

Vector quantization



Works better on images with more detail.

Conclusion

Overall, image 3 could be described as a matrix that consists of data being shown visually to the user as pixels of red, green and blue on a chosen device, such as a computer. The data can be

utilised by applying the SVD method to determine an estimate that is close to the original, but uses less storage on the disk. Using the SVD method grants 3(SVL)(r + c) information to be stored rather than 3(r × c), since the image was coloured.[2] To improve, further calculations on the compressed image quality could have been tested to confirm if the compressed image needed more refining, such as using the structural similarity index measure (SSIM). Another way this work could have been improved is by using a different compression technique, which would have given a better chance at decompressing back to the same image quality as the original starting image.

Image compression could be seen as an overall good way to reduce storage space on a drive or disk. This would allow for more images to be stored in one place at the same time.

References

Report

[1] Wikipedia contributors. (2021, December 25). *Image compression*. Wikipedia. Retrieved 5 January 2022, from https://en.wikipedia.org/wiki/Image_compression

[2] University of Utah, & Mathews, B. (2014, December). *Image Compression using Singular Value Decomposition (SVD)*. University of Utah. http://www.math.utah.edu/~goller/F15_M2270/BradyMathews_SVDImage.pdf

Source Code

- Trlin, G (2019) Using SVD for image compression in Python (Version 1.0) [Source code]. https://github.com/playandlearntocode/using-svd-for-image-compression-in-python-in-python

Appendix

SVD Code - Image Compression for Image 3

```
Key:
123abc = Own Code
123abc = 3rd Party Code, link in Source Code section



# Image is opened and 3 matrices are returned, with each matrix
corresponding respectively to the Red, Green and Blue channels.
Essentially, the colour image is being split into RGB channels.
def accessImage(imagePath):
    img3og = Image.open(imagePath)
```

```
    img3chdiv = numpy.array(img3og)

    R = img3chdiv[:, :, 0]
    G = img3chdiv[:, :, 1]
    B = img3chdiv[:, :, 2]

    return [R, G, B, img3og]


# The matrices of the Red, Green and Blue channels obtained from
accessImage are compressed separately, using the Singular Value
Decomposition method, and returned.
def compress(chDTAM, SVL):
    U, D, V = numpy.linalg.svd(chDTAM)
    chcmpr = numpy.zeros((chDTAM.shape[0], chDTAM.shape[1]))
    k = SVL

    LHS = numpy.matmul(U[:, 0:k], numpy.diag(D)[0:k, 0:k])
    chcmprin = numpy.matmul(LHS, V[0:k, :])
    chcmpr = chcmprin.astype('uint8')
    return chcmpr


# Image is called and sent to the accessImage function, along with the
colour variables.
R, G, B, oimg3 = accessImage('/content/drive/MyDrive/Image Analysis/Group
Coursework/images/IC_Assignment_Image3.bmp')

# This is the singular values limit, which determines the number of
singular values to use for reconstructing the compressed image.
SVL = 160

# The matrices for the Red, Green and Blue channels are displayed as plots
to show the colour distributions of the original image prior to
compression.
fig = plt.figure(figsize = (15,15))
fig.add_subplot(131)
plt.title("Blue Channel")
plt.imshow(B)
fig.add_subplot(132)
```

```
plt.title("Green Channel")
plt.imshow(G)
fig.add_subplot(133)
plt.title("Red Channel")
plt.imshow(R)
plt.show()
print('\n')

RC = compress(R, SVL)
GC = compress(G, SVL)
BC = compress(B, SVL)

# The compressed channel matrices for R, G and B are obtained and are
exported, creating the image memory needed for each channel.
imgR = Image.fromarray(RC, mode=None)
imgG = Image.fromarray(GC, mode=None)
imgB = Image.fromarray(BC, mode=None)

# The compressed image is finally constructed by merging the image memory
of the compressed channel matrices, R, G and B, together.
cimg3 = Image.merge("RGB", (imgR, imgG, imgB))

plt.figure(figsize = (30,30))
plt.title("Original Image")
plt.imshow(oimg3)
plt.show()

plt.figure(figsize = (30,30))
plt.title("Compressed Image")
plt.imshow(cimg3)
plt.show()

# The compression ratio for image 3 is calculated and displayed. The sizes
# are multiplied by 3 since we're dealing with colour images and the
# resulting values are in the form of bytes (divided by 1000000 to get
MB).

# The calculations use the original image's height and width, as well as
the compressed image's height and width.
oh = oimg3.height
```

```
ow = oimg3.width
cmprh = cimg3.height
cmprw = cimg3.width

ogSize = oh * ow * 3
cmprSize = SVL * (1 + cmprh + cmprw) * 3

print('Image 3 Original size: ' + str(ogSize) + ' bytes (' +
str(round(ogSize/1000000, 2)) + ' MB)\n')

print('Image 3 Compressed size: ' + str(cmprSize) + ' bytes (' +
str(round(cmprSize/1000000, 2)) + ' MB)\n')

rto = cmprSize / ogSize

print('Image 3 Compression ratio: ' + str(rto) + '\n')

print('Compressed image 3 size is ' + str(round(rto * 100, 2)) + '% of the
original image 3.')

# Final compressed image is saved to the directory.
 cimg3.save('/content/drive/MyDrive/Image Analysis/Group
Coursework/images/img3compressed.bmp')

# The Mean Square Error is calculated between the original image and the
compressed image.
dif = (np.subtract(oimg3, cimg3)) ** 2
mse = dif.mean()
mse

# The Peak Signal to Noise Ratio is calculated using the MSE value
obtained previously.
psnr = 10 * math.log10((1)**2/mse)
psnr
```

PCA and K-Means Code - Image Compression for Image 1

Dimensional reduction

```
from google.colab import drive
drive.mount('/content/drive')
```

```
imgbg = cv2.cvtColor(cv2.imread('/content/drive/MyDrive/image1.bmp'),
cv2.COLOR_BGR2RGB)
plt.imshow(imgbg)
plt.show()

#https://towardsdatascience.com/dimensionality-reduction-of-a-color-photo-
splitting-into-rgb-channels-using-pca-algorithm-in-python-ba01580a1118

#Splitting into the respective rgb channels
red,green,blue = cv2.split(imgbg)

#set the size of the figures
fig = plt.figure(figsize = (20, 9.5))

#Plot the image in each colour channel
#plot position on the console
fig.add_subplot(1,3,1)
plt.title("Red Channel")
plt.imshow(red)

fig.add_subplot(1,3,2)
plt.title("Green Channel")
plt.imshow(green)

fig.add_subplot(1,3,3)
plt.title("Blue Channel")
plt.imshow(blue)
#show figures
plt.show()

#make floating point number between 0-1
df_blue = blue/255
df_green = green/255
df_red = red/255

#reduce image from 485 dimensions to 50
pca_b = PCA(n_components=50)
pca_b.fit(df_blue)
trans_pca_b = pca_b.transform(df_blue)
```

```
pca_g = PCA(n_components=50)
pca_g.fit(df_green)
trans_pca_g = pca_g.transform(df_green)

pca_r = PCA(n_components=50)
pca_r.fit(df_red)
trans_pca_r = pca_r.transform(df_red)

#after dimensional reduction the channels must be inversed and then merged
back together to recreate the image
b_arr = pca_b.inverse_transform(trans_pca_b)
g_arr = pca_g.inverse_transform(trans_pca_g)
r_arr = pca_r.inverse_transform(trans_pca_r)

img_reduced= (cv2.merge((r_arr, g_arr, b_arr)))

#figure size
fig = plt.figure(figsize = (20, 9.5))
#original image
fig.add_subplot(121)
plt.title("Original Image")
plt.imshow(imgbg)

#reduced image
fig.add_subplot(122)
plt.title("Reduced Image")
plt.imshow(img_reduced)

plt.show()


#width,height = imbg.size
#nwidth,nheight = img_reduced.size

#orsize = width*height*3
#nsize = nwidth*nheight*3

#print('Image 3 Original size: ' + str(ogSize) + ' bytes (' +
str(round(ogSize/1000000, 2)) + ' MB)\n')
```

```
#print('Image 3 Compressed size: ' + str(cmprSize) + ' bytes (' +
str(round(cmprSize/1000000, 2)) + ' MB)\n')

imgbg.size

img_reduced.size

dif = (np.subtract(img, img_reduced)) ** 2
mse1 = dif.mean()
mse1

mse = np.mean((img - img_reduced) ** 2)
mse
```

Vector quantization

```
#https://stashable.blog/2019/12/09/image-compression-using-vector-quantiza
tion/

#vector quanitzation represents the pixels as vectors to store colour
informaiton and reduce the number of bits, here using k mean
img = cv2.cvtColor(cv2.imread('/content/drive/MyDrive/image1.bmp'),
cv2.COLOR_BGR2RGB)
plt.imshow(img)
plt.show()

#create numpty array from image
narray=np.array(img,dtype=np.float64)
origin=narray

#reshape the 3 dimensional array without changing its data,-1 will infer
the remaining length of the array from its dimensions
narray=narray.reshape((-1,1))

#create 4 clusters from the original data with the centre points of these
clusters being chosen at random
from sklearn.cluster import KMeans
kmeans=KMeans(n_clusters=4,init='random')
```

```
#pass array to clusters
kmeans.fit(narray)
#Center of clusters
centers=kmeans.cluster_centers_

#index for clusters
labels=kmeans.labels_

#create new array that uses the labels of the clusters which are just
indexes of the prediction clusters
f=[]
centers=centers.squeeze()

for i in labels:
  f.append(centers[i])
final=np.asarray(f)

#reshape the array to a 3 dimnesional array by using a copy of the
original
final=final.reshape(origin.shape)

#the array contains floating points numbers between 1-255, so divide to
get floating point numbers between 0-1
final=final/255

#display reduced image
plt.imshow(final)
plt.show()
```

Code - Image compression for image 2

```
import sys
from skimage import io
from sklearn.cluster import KMeans
import numpy as np

# reading filename

filename = sys.argv[1]

# reading the image
image = io.imread('library.png')

# preprocessing
rows, cols = image.shape[0], image.shape[1]
image = image.reshape(rows * cols, 3)

# modelling
print('Compressing...')
print('Note: This can take a while for a large image file.')
kMeans = KMeans(n_clusters = 16)
kMeans.fit(image)

# getting centers and labels
centers = np.asarray(kMeans.cluster_centers_, dtype=np.uint8)
labels = np.asarray(kMeans.labels_, dtype = np.uint8)
labels = np.reshape(labels, (rows, cols))
print('Almost done.')

# reconstructing the image
newImage = np.zeros((rows, cols, 3), dtype=np.uint8)
for i in range(rows):
    for j in range(cols):
            # assinging every pixel the rgb color of their label's center
            newImage[i, j, :] = centers[labels[i, j], :]
io.imsave('library.png'.split('.')[0] + '-compressed.png', newImage)

print('Image has been compressed sucessfully.')
```

Link to Google Colab Code:

https://colab.research.google.com/drive/1_NX94wBBmMkRokxshz1LWP99RsRzxLUN#scrollTo=eoRUvl0FM3m2

Google colab link code for image 2:

https://colab.research.google.com/drive/1b5FvovGhGHWxb_8MbqnKOYbqJKRZllzO#scrollTo=svhcYuamE0q0

Effort Allocation Sheet

Group: 3

Name	Contribution (%)	Note (briefly explain the contribution)	Signature
Shavin Croos	100	Did the code for the SVD compression and report writing	**T.S.Croos**
Uthman Maigari	100	Did the code for the K-Means compression and report writing	***bm.uthman***
Dan Keep	100	Did the code for the dimensional reduction and vector quantization and report writing	**D.Keep**