ASSIGNMENT 1
K-NEAREST NEIGHBOURS

CS-375 Machine Learning
Dr. Wajahat Hussain
Email: wajahat.hussain@gmail.com

Muhammad Usman
Registration Number 259861

Introduction

For this assignment, I had to implement the K-Nearest Neighbour algorithm in Google Colab and test it on my own photograph i.e., to check which actor/actress I resemble. The script was provided to us and I primarily had to write the code for calculating minimum distances, and finding their instances in the array of calculated distances. The procedure along with the python code is described below.

1. Mounting Google Drive and loading the dataset provided in 'data.mat' file

The first step in implementing this algorithm in Colab is to mount your drive. After that, provide the path of the folder in your drive that has the data.mat and test.jpg (the picture you want to run KNN on) files. Use the 'loadmat' command from the **SciPy** library for the extraction of data.mat file, it will return a python dictionary. Use the keys 'images' and 'C' to get the images and labels of the dataset, respectively. The code that implements this is given below:

```
# Enter your path of dataset from google drive
import scipy.io as sio
GOOGLE_COLAB = True

if GOOGLE_COLAB:
    from google.colab import drive, files
    drive.mount('/content/drive/')
    path = "/content/drive/MyDrive/ML KNN Classifier/"

dataset = path + "data.mat"

#Enter path of your test image
test_image_path = path + "test.jpg"

mat_contents = sio.loadmat(dataset)
images = mat_contents['images']
label = mat_contents['C']
```

The variables named 'images' and 'labels' are numpy arrays and their shape can be found using:

```
print(images.shape, label.shape)
```

Output: (50, 3072) (50, 1)

This indicates that there are 50 images in our dataset, each having 3072 pixels [32 (rows) x 32 (columns) x 3 (channels)] and a specific label.

2. Plotting the images in our dataset

You can use **pyplot** (from **matplotlib**) to view the images in your dataset. However, to use **pyplot.imshow** for your images, the input to imshow must be in the form (M, N, 3) instead of a row vector of length 3072.

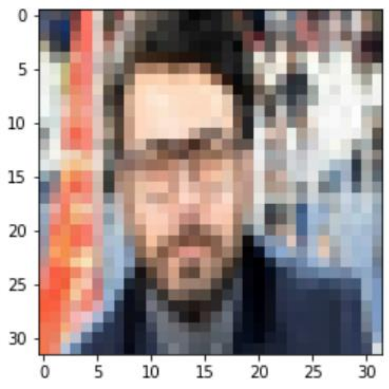
M	Height of the image in pixels
N	Width of the image in pixels
3	Colour channels

To reshape the images, **numpy.reshape** can be used. It takes, as input, the images array, the new shape (which must be compatible with the original array) and an additional argument 'order'. For your case, it is set to 'F' (Fortran-like index order) which means that the column-wise operations will be performed first.

```
import numpy as np
from matplotlib import pyplot as plt
import cv2

images = np.transpose(images)
print(images.shape)
im = np.reshape(images, [ 32, 32, 3, -1], order="F")
print(im.shape)

plt.imshow(im[:,:,:,44])
print(im[:,:,:,0].shape)
```

Output:

```
(3072, 50)

(32, 32, 3, 50)

(32, 32, 3)
```



Transpose interchanges the rows and columns of the images array, making its shape (3072, 50).

np. reshape converts the images array to (32, 32, 3, 50) shape. The 50 here is due to -1 as the fourth dimension whose value is automatically calculated by numpy and we do not have to specify it. 50 is the number of reference images in the dataset. The function **imshow** is then used to display the 44th image of our dataset in the code above.

3. Reading, resizing, and reshaping your image

To be able to use the K-Nearest Neighbours (KNN) algorithm on your image, you have to convert it in a format which KNN supports. You have already uploaded your test image on Google drive and stored its path in **`test_image_path`**. Use **cv2.imread** with your path as the argument to read your image. **`cv2.cvtColor`** and **`cv2.COLOR_BGR2RGB`** converts the default colour order (BGR) to the order (RGB) which is supported by matplotlib, so that the original-coloured image is displayed by **`plt.imshow`**.

After loading your image, resize it to 32 x 32 using cv2.resize and flatten it using the same np.reshape as discussed earlier with -1 as the dimension of the output image.

```
from scipy import misc
import cv2
from math import sqrt
from numpy import ndarray

for i in range(50):
    G = im[:,:,:,i]
    G = np.reshape(G,[-1], order="F")

    #Read your image here
####### Your code here #######

test_image = cv2.cvtColor(cv2.imread(test_image_path), cv2.COLOR_BGR2RGB)
print(type(test_image), test_image.shape)
plt.figure()
plt.imshow(test_image)

############################

    #Resize your image
####### Your code here #######

dim = (32, 32)
test_image = cv2.resize(test_image, dim)
print(type(test_image), test_image.shape)
plt.figure()
plt.imshow(test_image)

##############################

    #Reshape your image as we reshape the image of dataset
####### Your code here #######

T = np.reshape(test_image, [-1], order="F")
print(T.shape)

#############################
```

Output:

4. Calculating Euclidean distance between your test image and an image from the dataset

The Euclidean distance between two points in Euclidean n-space is given by

$$d(a,b) = \sqrt{\sum_{k=1}^{n} (b_k - a_k)^2}$$

You can implement this using a vectorized approach or a 'for' loop. First step is to change the data type of image arrays from 'uint8' (unsigned int 8) to int16, float32 etc. avoid overflow error in computation which may arise when a pixel having a higher value is subtracted from another having a lower value.

G and T are one dimensional arrays of `(3072,)`. `G.shape[0]` returns a scalar value of 3072 which is the stop value for the iterator i in `range(0, G.shape[0])`. Note that this stop value is exclusive, meaning that the final value of i will be 3071 instead of 3072.

In the loop, for one iteration, you take the difference of single data point in T (test image) and G (single dataset image), use `pow` to square the difference and add the result to sum_result. After the execution of this loop, take square root of sum_result using sqrt (from math) to get a scalar value i.e., distance between these two images.

```
sum_result = 0
T = T.astype('int16')
G = G.astype('int16')

for i in range(0, G.shape[0]):
  x = pow((T[i] - G[i]), 2)
  sum_result =  sum_result + x

d = sqrt(sum_result)
print('Distance d = ', d)
```

5. Writing the code for 1-NN

After completing the above steps, you are now ready to write the code for KNN. We will start with K = 1. First step is to calculate the distance between your test image and each image of the dataset and store the result in the list `distance_arr`. This part of KNN is similar to the code written for Euclidean distance calculation in the previous step. However, we are now finding the distance of the test image with each image in the dataset and appending the result to our list `distance_arr`.

```
distance_arr = []
for i in range(50):
  G = im[:,:,:,i]
  G = np.reshape(G,[-1], order="F")

  sum_result = 0;

  for i in range(0, G.shape[0]):
    x = (T[i] - G[i])**2
    sum_result =  sum_result + x

  distance_arr.append(sqrt(sum_result))
```

After getting the list of distances (`distance_arr`), you need to convert it to a numpy array so that you can use functions such as min (minimum value of an array) etc. on your calculated distances. Use `distance_min = min(distance_arr)` to get the minimum distance value from your array of calculated distances.

```
distance_arr = np.array(distance_arr)
distance_min = min(distance_arr)
```

Now that you have the minimum value of distance, you need to find the index at which it is occurring and then display the image and label of that index. `index_condition` is the condition which **np.where** (numpy function that returns the indices of elements in an input array where the given condition is satisfied.) will use to find the index in your array of distances `distance_arr`.

Set the condition as shown below to get the index i.e., image giving the minimum distance value. You have now successfully implemented KNN for K = 1.

Now you need to display the image and label, corresponding to that index number. Note that np.where returns a ndarray instead of a scalar value. So, to obtain a scalar value, use a for loop as shown below.

```
index_condition = distance_arr == distance_min
distance_min_index = np.where(index_condition)[0]

for data in distance_min_index:
  print(label[data][0])
  plt.imshow(im[:,:,:, data])
```

Output:

```
['tom cruise']
```

6. Writing the code for 3-NN

Once you have written the code for 1-NN, KNNs for different values of K are obtained just by slightly modifying the code for 1-NN. Repeat the previous process for 1-NN until you have obtained the array `distance_arr`. Make another copy of this array and sort it using **np.sort**. This will sort the array in ascending order. Next, take the first three elements of that array (K elements for KNN) and store it in another variable (`d_min`).

```
K = 3
distance_arr = []  #-- Array of calculated Distances
for i in range(50):
  G = im[:,:,:,i]
  G = np.reshape(G,[-1], order="F")

  sum_result = 0;
  for i in range(0, G.shape[0]):
    x = (T[i] - G[i])**2
    sum_result =  sum_result + x

  distance_arr.append(sqrt(sum_result))

distance_arr = np.array(distance_arr)
distance_arr_sorted = np.array(distance_arr)
distance_arr_sorted.sort()
d_min = distance_arr_sorted[0: K]
```

Now, we want to find the indices where the first three minimum distances occur in our original distance array. Same concept of condition and '**np.where**' as done in 1-NN is used here but a loop is applied which changes the condition to the next minimum distance value. The rest is same for 3-NN classifier.

```
d_min_indices = []
for i in range(K):
  index_condition = (distance_arr == d_min[i])
  distance_min_index = np.where(index_condition)[0]
  d_min_indices.append(distance_min_index)

for data in d_min_indices:
  plt.figure()
  plt.imshow(im[:,:,:, data[0]])
  plt.title((label[data][0][0]))
```

Variable d_arr contains the computed distances of the reference dataset images with the test image; my photograph.

```
d_arr:
array([4780.29130911, 7358.12428544, 5854.33728103, 4441.93989153,
       6251.13253739, 5209.44603581, 5939.82491324, 7214.55979253,
       5768.00476768, 5699.2938159 , 4311.04940821, 5507.47927822,
       6168.18085014, 5632.88416355, 6728.0739443 , 5483.22596653,
       5866.38977566, 4865.24912003, 4792.90913747, 5394.71556247,
       5574.4996188 , 6642.67845075, 8094.9092027 , 5578.63701992,
       5470.80304891, 5357.51285579, 6286.53457797, 5437.55799969,
       5517.00725031, 5498.04874478, 4716.06382484, 6712.36925385,
       5432.29003276, 6063.82008308, 6352.87375917, 5618.56173767,
       5823.43979105, 6498.18466958, 5440.65474001, 5974.21701983,
       5347.54644674, 5419.84566201, 5136.34403053, 7614.68489171,
       5290.31549154, 7451.258283  , 5975.67017162, 5135.05102214,
       4826.67483885, 6322.28226197])
```

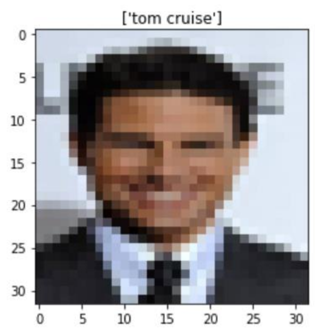
Variable d_min indicates the first three minimum distances from the d_arr variable.

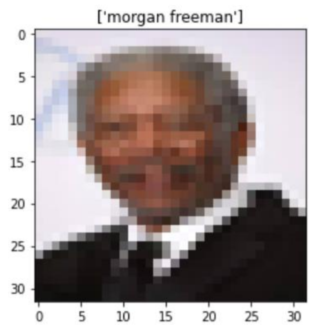
```
d_min: array([4311.04940821, 4441.93989153, 4716.06382484])
```

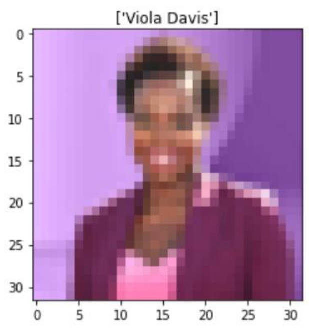
The following variable indicates the indices of the corresponding minimum distance reference images.

```
d_min_indices: [array([10]), array([3]), array([30])]
```

Output:







As mentioned earlier, `d_min_indices` return a ndarray instead of a scalar. So, data in `d_min_indices` (for loop) gives us `array([10])` which is converted to a scalar as `data[0]`. You can now implement KNN for other values of K just by changing the value of K in the above code.

Summary

In this tutorial, we've utilized the k-nearest neighbours algorithm to find our lookalike from a dataset of reference images. The images were provided in a matlab data file (.mat), and were imported using the SciPy library. Preliminary processing of the images from a multidimensional array to a singular input vector is done using the numpy library. For the algorithm implementation we utilized the numpy and the math libraries. We then imported our own photograph using OpenCV. For evaluation and visualization, we use the pyplot module of matplotlib.