Classify faces using unsupervised approach.

Objective:

To use unsupervised approach and cluster the images/faces based on how similar the feature vectors are.

The model:

The pre-trained model that will be used in this tutorial is the VGG16 convolutional neural network (CNN), which is considered to be state of the art for image recognition tasks. We are going to be using this model as a feature extractor only, meaning that we will remove the final (prediction) layer so that we can obtain a feature vector.

The Data:

This implementation will use the faces dataset from I'm Beside You. The dataset contains 57898 images of 4 different faces of people (including noise) in jpg format.

https://drive.google.com/file/d/1wSW18_sQzlQ7xN3Y41UhfOlmCL8dhu-J/view?usp=sharing consists of face images extracted from three different zoom meeting videos.
https://drive.google.com/file/d/1RnsgIfAqpbjjDWSXS55w_jRBCBEivaz4/view?usp=sharing consists of annotations.

Imports:

Before we get started, we need to import the modules needed in order to load/process the images along with the modules to extract and cluster our feature vectors.

- load_img allows us to load an image from a file as a PIL object
- img_to_array allows us to convert the PIL object into a NumPy array
- preproccess_input is meant to prepare your image into the format the model requires. You should load images with the Keras load_img function so that you guarantee the images you load are compatible with the preprocess_input function.
- VGG16 is the pre-trained model we're going to use
- KMeans the clustering algorithm we're going to use
- PCA for reducing the dimensions of our feature vector

```
# for loading/processing the images
from tensorflow.keras.utils import load_img
from tensorflow.keras.utils import img_to_array
from keras.applications.vgg16 import preprocess_input

# models
from keras.applications.vgg16 import VGG16
from keras.models import Model
from sklearn.cluster import KMeans

# clustering and dimension reduction
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split

# for everything else
import os
import numpy as np
import matplotlib.pyplot as plt
from random import randint
import pandas as pd
import pickle
```

Loading the data and preprocessing(in excel file):

- Load the excel file
- Check for NaN values
- Remove Noises(e.g faces with mask and non_faces)

```
import pandas as pd
df=pd.read_excel("(English translate)20210426_image_classification_Imbesideyou (1).xlsx")
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 30082 entries, 1 to 57898
Data columns (total 5 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   Nan        0 non-null      object
 1   No         30082 non-null  object
 2   File Name  30082 non-null  object
 3   ID         30082 non-null  object
 4   Mask       30082 non-null  object
dtypes: object(5)
memory usage: 1.4+ MB
```

Removing non_faces and mask images

```
In [7]: df = df.drop(df.index[df['Mask'] == 'Y'])
        df = df.drop(df.index[df['ID'] == 0.0])
```

```
In [8]: df
```

	Nan	No	File Name	ID	Mask
1	NaN	1.0	187dcd69-e620-46b5-b40e-78020096caf4_00000850_...	1.0	N
2	NaN	2.0	187dcd69-e620-46b5-b40e-78020096caf4_00000851_...	1.0	N
3	NaN	3.0	187dcd69-e620-46b5-b40e-78020096caf4_00000852_...	1.0	N
4	NaN	4.0	187dcd69-e620-46b5-b40e-78020096caf4_00000853_...	1.0	N
5	NaN	5.0	187dcd69-e620-46b5-b40e-78020096caf4_00000854_...	1.0	N
...	...	...	...	...	...
57892	NaN	57892.0	ac0abb38-d3ff-400a-b710-7bdcbb7c366d_00003941_...	1.0	N
57893	NaN	57893.0	ac0abb38-d3ff-400a-b710-7bdcbb7c366d_00003941_...	4.0	N
57894	NaN	57894.0	ac0abb38-d3ff-400a-b710-7bdcbb7c366d_00003941_...	2.0	N
57897	NaN	57897.0	ac0abb38-d3ff-400a-b710-7bdcbb7c366d_00003942_...	1.0	N
57898	NaN	57898.0	ac0abb38-d3ff-400a-b710-7bdcbb7c366d_00003942_...	4.0	N

30082 rows × 5 columns

Loading the data:

we want python to point to the location where the images are located. This way instead of loading a whole file path, we can simply just use the name of the file.

- Mingle file contains all the images(faces) of the 3 folders

```
File_name=df['File Name'].tolist()
```

```
len(File_name)
```

```
30082
```

Locating path and creating a list of all valid faces for training

```
path = r"C:\Users\beher\jupyterZ\imbesideyou\face_images\face_images\mingle"
# change the working directory to the path where the images are located
os.chdir(path)

# this list holds all the image filename
faces = []

# creates a ScandirIterator aliased as files
with os.scandir(path) as files:
  # loops through each file in the directory
    for file in files:
        if file.name.endswith('.jpg') and file.name in File_name  :
          #adds only the image files to the flowers list
            faces.append(file.name)
```

```
print(faces[:10])
```

```
['187dcd69-e620-46b5-b40e-78020096caf4_00000850_0.jpg', '187dcd69-e620-46b5-b40e-78020096caf4_00000851_0.jpg', '187dcd69-e620-46b5-b40e-78020096caf4_00000852_0.jpg', '187dcd69-e620-46b5-b40e-78020096caf4_00000853_0.jpg', '187dcd69-e620-46b5-b40e-78020096caf4_00000854_0.jpg', '187dcd69-e620-46b5-b40e-78020096caf4_00000855_0.jpg', '187dcd69-e620-46b5-b40e-78020096caf4_00000856_0.jpg', '187dcd69-e620-46b5-b40e-78020096caf4_00000857_0.jpg', '187dcd69-e620-46b5-b40e-78020096caf4_00000858_0.jpg', '187dcd69-e620-46b5-b40e-78020096caf4_00000859_0.jpg']
```

Data Preprocessing:

This is where we put the load_img() and preprocess_input() methods to use. When loading the images we are going to set the target size to (224, 224) because the VGG model expects the images it receives to be 224x224 NumPy arrays.

```
# load the image as a 224x224 array
img = load_img(faces[0], target_size=(224,224))
# convert from 'PIL.Image.Image' to numpy array
img = np.array(img)

print(img.shape)
```

```
(224, 224, 3)
```

Currently, our array has only 3 dimensions (rows, columns, channels) and the model operates in batches of samples. So we need to expand our array to add the dimension that will let the model know how many images we are giving it (num_of_samples, rows, columns, channels).

```
reshaped_img = img.reshape(1,224,224,3)
print(reshaped_img.shape)
```

```
(1, 224, 224, 3)
```

The last step is to pass the reshaped array to the preprocess_input method and our image is ready to be loaded into the model.

```
x = preprocess_input(reshaped_img)
```

The Model:

Now we can load the VGG model and remove the output layer manually. This means that the new final layer is a fully-connected layer with 4,096 output nodes. This vector of 4,096 numbers is the feature vector that we will use to cluster the images.

```
# load model
model = VGG16()
# remove the output layer
model = Model(inputs=model.inputs, outputs=model.layers[-2].output)
```

Function for Feature Extraction:

defining function for feature extraction

```
# load the model first and pass as an argument
model = VGG16()
model = Model(inputs = model.inputs, outputs = model.layers[-2].output)

def extract_features(file, model):
    # load the image as a 224x224 array
    img = load_img(file, target_size=(224,224))
    # convert from 'PIL.Image.Image' to numpy array
    img = np.array(img)
    # reshape the data for the model reshape(num_of_samples, dim 1, dim 2, channels)
    reshaped_img = img.reshape(1,224,224,3)
    # prepare image for model
    imgx = preprocess_input(reshaped_img)
    # get the feature vector
    features = model.predict(imgx, use_multiprocessing=True)
    return features
```

Feature Extraction of all valid faces:

feature extraction for all valid faces

```
data = {}
p = r"C:\Users\beher\jupyterZ\imbesideyou\needed"

# lop through each image in the dataset
for face in faces:
    # try to extract the features and update the dictionary
    try:
        feat = extract_features(face,model)
        data[face] = feat
    # if something fails, save the extracted features as a pickle file (optional)
    except:
        with open(p,'wb') as file:
            pickle.dump(data,file)


# get a list of the filenames
filenames = np.array(list(data.keys()))

# get a list of just the features
feat = np.array(list(data.values()))

# reshape so that there are 210 samples of 4096 vectors
feat = feat.reshape(-1,4096)
```

```
1/1 [==============================] - 1s 709ms/step
1/1 [==============================] - 0s 189ms/step
1/1 [==============================] - 0s 191ms/step
1/1 [==============================] - 0s 183ms/step
1/1 [==============================] - 0s 205ms/step
1/1 [==============================] - 0s 218ms/step
1/1 [==============================] - 0s 169ms/step
1/1 [==============================] - 0s 174ms/step
1/1 [==============================] - 0s 176ms/step
```

Dimensionality Reduction (PCA):

- Since our feature vector has over 4,000 dimensions. We can't simply just shorten the list by slicing it or using some subset of it because we will lose information.
- If only there was a way to reduce the dimensionality while keeping as much information as possible. Enter the realm of principle component analysis.
- working with data and have a lot of variables to consider (in our case 4096), PCA allows us to reduce the number of variables while preserving as much information from the original set as possible.
- The number of dimensions to reduce down to is up to you and I'm sure there's a method for finding the best number of components to use, but for this case, I just chose 100 as an arbitrary number.

Dimesnional reduction

```
pca = PCA(n_components=100, random_state=22)
pca.fit(feat)
x = pca.transform(feat)
```

KMeans clustering:

This algorithm will allow us to group our feature vectors into k clusters. Each cluster should contain faces that are visually similar. In this case, we know there are 4 different faces of people so we can have k = 4 (after removing noise)

creating unique labels for Kmeans clustering

```
# get the unique labels
label = df['ID'].tolist()
unique_labels = list(set(label))
```

Kmeans clustering fitting(training)

```
kmeans = KMeans(n_clusters=len(unique_labels), random_state=22)
kmeans.fit(x)
```

```
KMeans(n_clusters=4, random_state=22)
```

```
kmeans.predict(x)
```

```
array([3, 3, 3, ..., 1, 3, 1])
```

- By default Kmeans clustering creates clusters and names them as cluster 0,1,2,3 etc.
- So we have 4 clusters as 0,1,2,3

Storing all the filenames and kmeans cluster labels in a dictionary.

storing all the labels and filenames in a dictionary

```
groups = {}
for file, cluster in zip(filenames,kmeans.labels_):
    if cluster not in groups.keys():
        groups[cluster] = []
        groups[cluster].append(file)
    else:
        groups[cluster].append(file)
```

Function for viewing the clusters:

function for viewing clusters

```
def view_cluster(cluster):
    plt.figure(figsize = (25,25));
    # gets the list of filenames for a cluster
    files = groups[cluster]
    # only allow up to 100 images to be shown at a time
    if len(files) > 100:
        print(f"Clipping cluster size from {len(files)} to 100")
        files = files[:100]
    # plot each image in the cluster
    for index, file in enumerate(files):
        plt.subplot(10,10,index+1);
        img = load_img(file)
        img = np.array(img)
        plt.imshow(img)
        plt.axis('off')
```

Viewing Clusters:

- Cluster_0

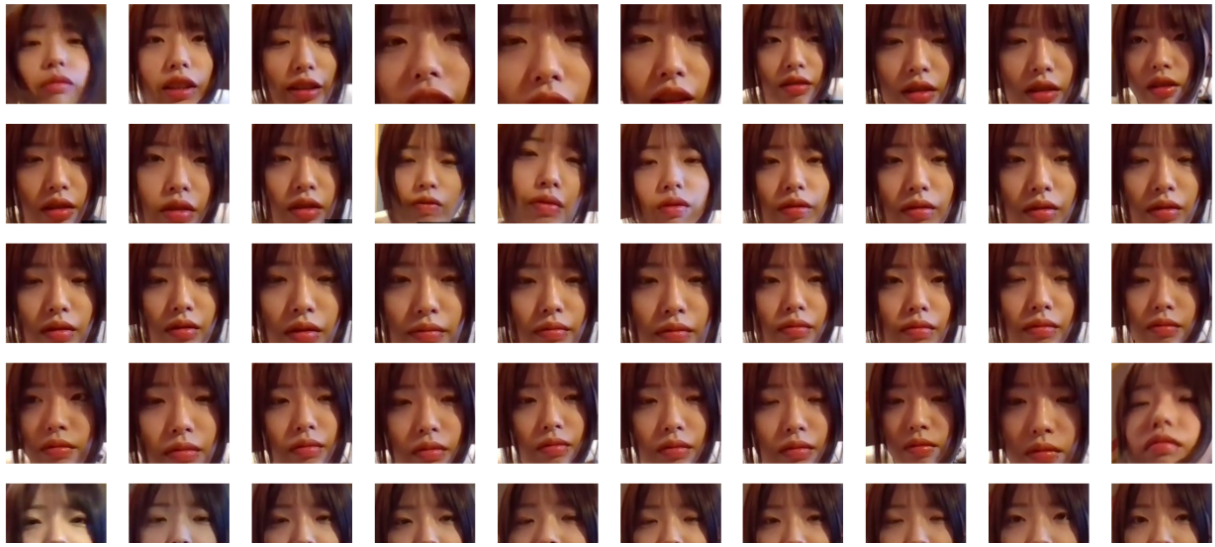
```
view_cluster(0)
```

```
Clipping cluster size from 4720 to 30
```

Cluster_1

```
view_cluster(1)
```

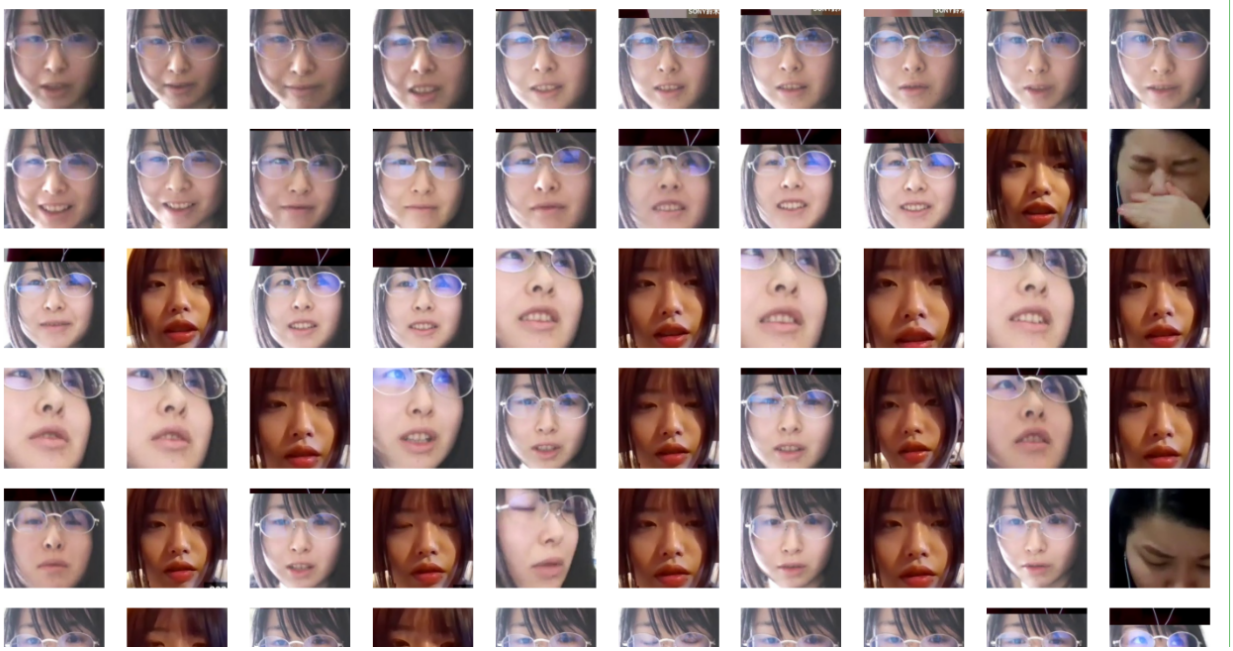
```
Clipping cluster size from 10541 to 100
```



Cluster_2

```
view_cluster(2)
```

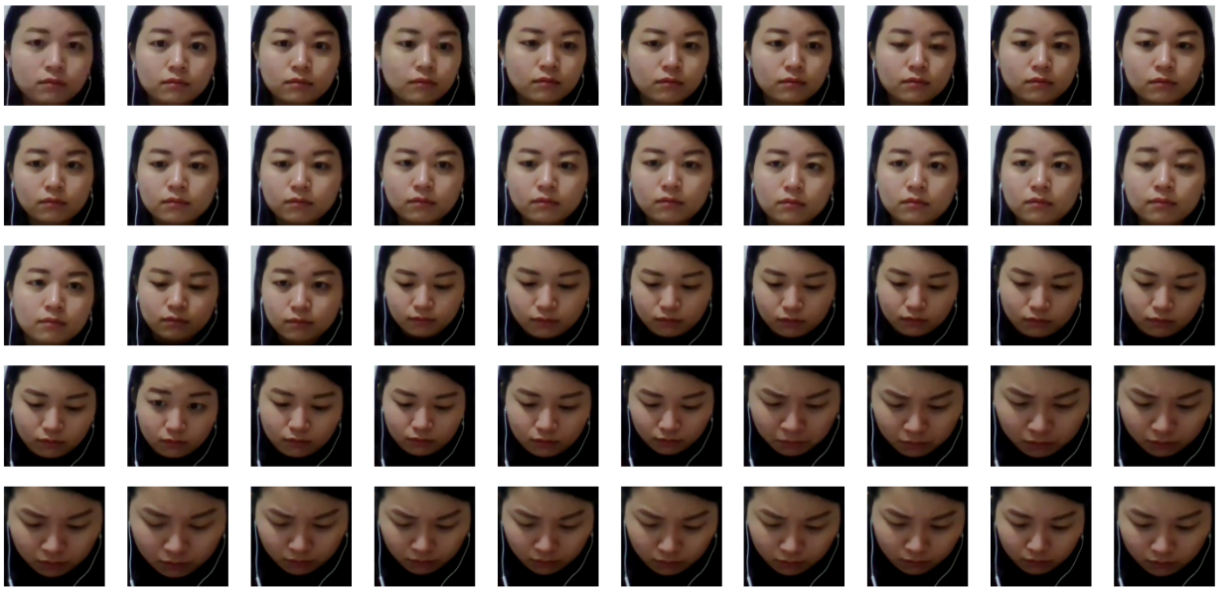
```
Clipping cluster size from 6004 to 100
```



Cluster_3

```
view_cluster(3)
```

```
Clipping cluster size from 8817 to 100
```



Testing and accuracy

- As we know that we know the cluster names created by Kmeans are not same as label(ID) present in annotation file ,we have to create a new array with a little bit of modifications with Kmeans.predict(x) array

By reference to the images and the excel file we can predict that the clusters are ¶

ID->cluster,

1->3 ,

4->0 ,

3->2 ,

2->1

- We need to run a loop to make required modifications and create a new list

```
y_pred=[]
```

```
for i in range (len(kmeans.predict(x))):
    print(i)
    if kmeans.predict(x)[i]==3:
        y_pred.append(1.0)
        continue
    elif kmeans.predict(x)[i]==0:
        y_pred.append(4.0)
        continue
    elif kmeans.predict(x)[i]==2:
        y_pred.append(3.0)
        continue
    elif kmeans.predict(x)[i]==1:
        y_pred.append(2.0)
        continue
```

Accuracy

```
y_pred=np.array(y_pred)
```

```
y_test=df["ID"].to_numpy()
y_test= y_test.astype('float64')
```

```
metrics.accuracy_score(y_test,y_pred)
```

```
0.9033641380227379
```

```
f1_score(y_test,y_pred, average='weighted')
```

```
0.9003582031604779
```

- Metrics used are accuracy_score and f1_score and above are their their scores.