CS156 Final Project
December 2020
Akmarzhan Abylay

Problem Description

With BTS's and BlackPink's rising popularity in the last few years, K-pop has become a global phenomenon. It has gathered millions of fans and record-breaking achievements (e.g., BTS has more than 40 million official fans, and their song Dynamite has topped the Billboard Artist 100 chart for 13 consecutive weeks) (Billboard, 2020). As more people get interested in Korean pop music, the number of those who get confused by the similarity in the looks of k-pop idols rises as well. Unless you are an experienced k-pop stan or just really good at recognizing people, telling idols apart, especially in MVs (Music Videos), is difficult. I am an A.R.M.Y. (i.e., the name for BTS fans) myself and I observed that many new fans can't recognize all members. For example, they cannot differentiate V and Jungkook (JK). The below image is a merge of them. Can you tell the difference?

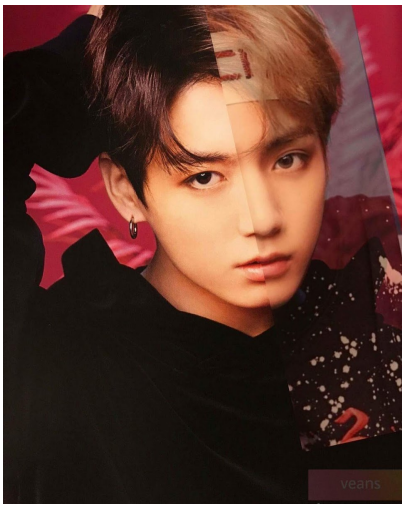

Figure 1: Jungkook and V merged (borrowed from here)

I was interested in the reasons behind, so I dug deeper and found out that there is a phenomenon called the "Cross-race identification bias." It says people of one race find it challenging to recognize the facial features and expressions of people of another race. That made sense because most of the fans are international so they are not used to seeing Korean faces a lot. Although I am Asian and I am used to seeing Korean faces (i.e., I became interested in Korean culture in 2013), I sometimes find it hard to memorize the faces within the new k-pop groups. This is also because they often have to go through plastic surgeries to fit Korean beauty standards and look alike (Stone, 2013).

This is why it was interesting for me to try building an algorithm that could tell the members apart and indicate their names so that it would be helpful to fans. I will create two versions of the face recognition algorithm: one using the pre-existing libraries, and the other one by fine-tuning the VGGFace model created by the Visual Geometry Group (VGG) at Oxford.

Solution Specification

Optimization

As mentioned, I aimed to use two methods: one with the existing package that detects and recognizes the faces automatically and the other through fine-tuning the VGGFace model. I explained them below.

Face Recognition Library

For this approach, I used the 'face_recognition' library (see Documentation here) for the face detecting and encoding steps. For processing the videos, I used the cv2 library.

We would need to consider the following:

1. **KNOWN FACES:** since we have some faces that need to appear in a particular video (i.e., BTS members in our case), the newly detected faces need to be compared with the known ones. To do that, we need to encode the known faces (i.e., return the most important features in a face that would differentiate that specific person), label them correspondingly and save them. The encoding depends on how the model was trained and what it thinks is crucial for the face. The face_recognition library automatically does it for you (i.e., face_encodings method) based on their pre-trained (i.e., on adults) model.
2. **PROCESS VIDEO:** Now, we need to process the video frame by frame since we would detect and recognize the faces on each picture. I will use cv2 to write the input video and then add rectangles around the faces with the corresponding members' names. Then, the final frames will be written to a new file and saved on the computer.
3. **FACE DETECTION:** The problem of face detection is understanding where the face is on a given frame. The face_recongition library does this through the command face_locations, which returns an array of boxes (coordinates) of faces in an image. It uses dlib for these purposes.
4. **FACE RECOGNITION:** Based on the new face that we detected in the previous step, it encodes it in the same fashion as described above. There are different methods for encoding, from which the library provides:
   (a) **face_distance:** calculates the euclidean distance as a means of comparison between the given face encodings and the known face encodings. The distance basically serves as a metric of how similar the faces are (i.e., the smaller the value, the closer they are).
   (b) **compare_faces**: compares the given face encodings with the known face encodings to see if they match. Returns True/False based on the given tolerance, so it is similar to the face_distance method but includes a tolerance.

I will combine both of them and choose the match that a) passes the tolerance for distance and b) has minimal distance among all encodings.

The input would be the video with BTS members, while the output is the processed video with the names. This approach is good because of:

- **convenience**: It has all of the methods for detecting and recognizing the faces built-in, so it is very convenient since we don't have to build models from scratch.
- **performance**: It has been extensively trained on thousands of pictures (i.e., the documentation doesn't specify how many explicitly) of adults, which is why it will perform better than a model that was trained on a few samples. Also, it requires only a few data points (e.g., around ten pictures for each class were enough for it to be able to recognize members relatively well).

This approach has some drawbacks, too, though:

- **not flexible**: The model for extracting the features cannot be changed (as far as I understand), which is why there is no way to tweak the model (e.g., change the layers in the model) or make it better.

VGGFace Fine Tuning

We could do feature extraction, but since the above method is already doing it, I decided to try something different. In the previous part, we extracted the image features using a pre-existing model and then compared these encodings to the library of known encodings. For this step, I performed fine-tuning (i.e., unfreezing the weights and performing a few more iterations of gradient descent) to help the network specialize in classifying the seven members.

I didn't change the original VGGFace architecture much, just deleting the top and adding an extra layer to tailor it to classifying members. Also, I used ImageDataGenerator to automatically import the images and corresponding classes from the directories. I used a sparse categorical cross-entropy loss function, which is the same as categorical cross-entropy but for indexed labels (i.e., not one-hot encoded). This is a suitable loss function because our classes are mutually exclusive. I first compiled the model as usual (i.e., frozen weights) and then performed a few more iterations with a small learning rate with the weights unfrozen.

The training accuracy was 69.3% (192 pictures), and the testing accuracy was 48.14% (27 photos), which indicates that the model was overfitting (i.e., fit too closely on the training set). This might be because I had very few pictures for the testing set and the neural network architecture was not suitable. I tried to improve the performance by adding extra Dropout layers and changing the complexity of the architecture, but it didn't seem to help that much. This is why I ended up adding more data, which helps in most cases since the model will have more instances of the members and take away more important features. **Note**: Both the VGGFace and the Face Recognition library were trained on the same amount of data.

The rest of the algorithm was quite similar to the previous model since I analogously used each frame, detected faces in it, cropped the face, and fed the picture into the neural network so that it classifies the member.

Once again, the input would be the video with BTS members, while the output is the classified video. This approach is good because of:

- **flexibility**: I could change the model and the network architecture depending on what I need, as well as train the model on the data I have. There is a lot of room for improvement and experimentation.
- **relevance**: The VGGFace model was pre-trained on thousands of faces already, so it is a good model to start the face recognition exploration.

However, it has its own disadvantages:

- **performance**: It doesn't perform as well as the existing package, although we use the pre-trained VGGFace model.
- **time**: It takes more time than the face_recognition library to compile.

Testing and Analysis

I tested one video with labels and one video without tags. The original videos were taken from BTS's "Black Swan" and "Boy With Luv" music videos. Below are some screenshots from the two models for the labeled video (see Appendix A for code for Face Recognition library and Appendix B for VGGFace).

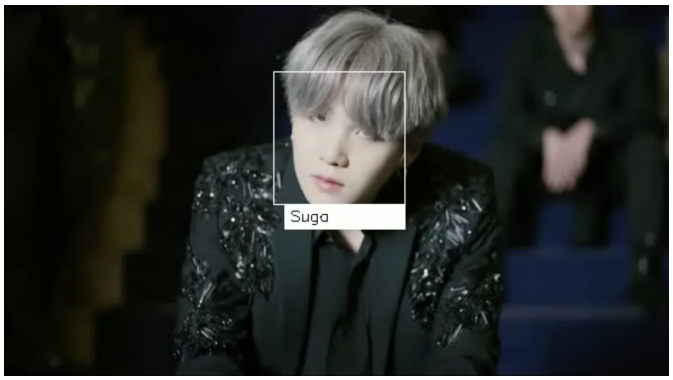

Figure 2: (Correct) Face Recognition model screenshot (link to video)

Figure 3: (Incorrect) VGGFace model screenshot (link to video)

I used the same metric to assess the performance of the models. I used the **accuracy score** (i.e., ratio of the correctly identified labels over the total number of labels). Since I was working with videos, I considered each frame as a test data point, so for the video I tested, I got 479 points since it had 479 frames. I manually labeled the videos and then checked whether the members are correctly classified. This is a good metric as the models have never seen the frames before.

Face Recognition Library

I got an accuracy of approximately 80.2%, which is pretty good given that the model didn't know what members looked like in the music video specifically. Also, each member had only around 25-30 photos each (a total of 192 photos for 7 categories), so I think the performance is relatively good.

An interesting observation about the library model is that the biggest thing it was confused about is whether V was JK or not. As mentioned at the beginning of the assignment, new fans find it hard to recognize these two (i.e., it becomes relatively easy after some time).

VGGFace

For the neural network model, the accuracy score isn't as high, with 65%. This is because we had a small dataset, 7 categories, and BTS members mostly had the same makeup and lighting, so the model confused members a lot.

More testing

I wanted to label the second video as well using a similar technique. I would have had labels for each frame, and if there are two or more people, then based on which one is ordered first, I would have had multiple labels for each frame. I would then count each label-prediction pair as an observation so that I would have more data points. I wanted to test the second video since the first one showed each member separately, and they are not moving around a lot. However, the second video had a lot more movements (i.e., mainly dancing), and it ended up taking me a lot more time than I initially thought. This is why I decided to keep only one video labeled (i.e., it is easier and still has enough frames for a fair estimate) and visually inspect the second video results.

From visual inspection, the face_recognition library is doing well. It misclassified only a few people and for a small period while recognizing JK, Jimin, and J-Hope the best. The VGG16, on the other hand, is not doing as great, constantly mixing some of the members.

Conclusions

In this assignment, I tried to create a tool that would differentiate a South-Korean boy group BTS. Both models worked, although the VGGFace one didn't perform as well as the face_recognition library. In any case, there is still room for improvement, starting from the method I chose for classification to labeling and testing videos. We could also get more data since thousands of k-pop idol photos are available online, although there are no sorted, labeled datasets that could easily be downloaded. These models could be applied for classifying other k-pop groups as well.

I am interested in this topic, which is why I might explore it in more detail for my Capstone. I am thinking about something like a Google Chrome extension that would automatically apply it to videos. Although there are many more complications (e.g., time, accuracy) that need to be fixed, I feel like this idea is very relevant, especially given the increasing interest in k-pop.

References

1. Big Hit Labels. (2020). BTS () 'Black Swan' Official MV. [Video File]. Retrieved from https://www.youtube.com/watch?v=0lapF4DQPKQ
2. Big Hit Labels. (2019). BTS () ' (Boy With Luv) (feat. Halsey)' Official MV. [Video File]. Retrieved from https://www.youtube.com/watch?v=XsX3ATc3FbA
3. Billboard. (2020). BTS Becomes First Group to Rule Artist 100, Hot 100 Billboard 200 Charts at the Same Time. Retrieved from https://www.billboard.com/articles/business/chart-beat/9492020/bts-first-group-rule-artist-100-hot-100-billboard-200-charts/: :text=BTS
4. Geitgey, A. (2020). Face Recognition Documentation. Retrieved from https://buildmedia.readthedocs.org/media/pdf/face-recognition/latest/face-recognition.pdf
5. Stone, Z. (2013). The K-Pop Plastic Surgery Obsession. Retrieved from https://www.theatlantic.com/health/archive/2013/05/the-k-pop-plastic-surgery-obsession/276215/)

LOs

- **#neuralnetworks:** used a neural network appropriate to the context (i.e., fine-tuning a VGGFace model which was pre-trained on thousands of faces); provided a clear explanation supporting the choice (e.g., provides flexibility); accurately used a neural network in the given context and explained the limitations of the approach, as well as contrasted it with another method.
- **#modelmetrics:** applied the correct metrics for assessing the performance in the relevant context (i.e., sparse categorical cross-entropy for the loss function and the accuracy for evaluating the video performance) and provided an explanation for the choice.
- **#overfitting:** used an effective preventative overfitting strategy appropriate for the context (i.e., adding more data points); justified the choice and talked about how the competing alternatives didn't solve the problem (e.g., changing the complexity of the neural network wasn't very effective in comparison).

Appendix

1 Appendix A: Face Recognition Library

[1]:
```
#importing the needed libraries
import face_recognition
import cv2
import os
import numpy as np
```

[2]:
```
def encoding(face_dir):
    """
    Function for encoding the 'known' faces to then store them and use for␣
 ↪comparing.

    Input:
        face_dir: a directory of known faces with each person under their own␣
 ↪folder.

    Output:
        faces: list with the encoded faces (i.e., noted important features and etc.
               from the provided sample).
        names: list with the names of the people who are encoded (i.e., corresponds
               to the 'faces' list).
    """

    #initializing
    faces, names = [], []

    #looping through each person's folder
    for name in os.listdir(face_dir):

        #os.listdir finds some documents that we don't need, so this is the easiest␣
 ↪way
        #to ignore them
        if name == '.DS_Store':
            continue
        for file in os.listdir(f'{face_dir}/{name}'):
            if file == '.DS_Store':
                continue

            #loading images (one person at a time)
            image = face_recognition.load_image_file(f'{face_dir}/{name}/{file}')

            #some of the pictures I found weren't pictures but rather .webp, and I␣
 ↪didn't
            #want to go through all of them again so I used the below line
            if len(face_recognition.face_encodings(image))!=0:
```

```
                #actual encodings
                encoding = face_recognition.face_encodings(image)[0]

                #storing the encodings of the faces and the corresponding names
                faces.append(encoding)
                names.append(name)
    #output
    return faces, names

def in_out(inp, out):
    """
    Function for reading the input video and initializing the output video.

    Input:
        inp: path to the input video that we want to process.

    Output:
        input_mov: VideoCapture object containing the input video file.
        output_mov: VideoWriter object which will store the output video.
    """

    #setting the input video we want to process
    input_mov = cv2.VideoCapture(inp)

    #setting the output video to match the size of the input video and have a
    #specific name and format
    fourcc = cv2.VideoWriter_fourcc('m', 'p', '4', 'v')
    output_mov = cv2.VideoWriter(out, fourcc, 20.0, (int(input_mov.
 ↪get(3)),int(input_mov.get(4))))

    #output
    return input_mov, output_mov
```

[3]:
```
def clf(inp, out, tol, model, test, prev_acc=None):
    """
    Function for reading the input video and initializing the output video.

    Input:
        inp: path to the input video that we want to process.
        out: path to the output video.
        tol: path to the input video that we want to process.
        model: the model to use for face detection.
        test: the test set to compare results.
        prev_acc (default = None): previous accuracy list to
                                   keep track of results.

    Output:
        accuracy: the list with 0/1 depending on whether the
        classification is correct.
    """
```

```
    input_mov, output_mov = in_out(inp, out)

    #seeing whether we should keep track of some previous accuracy
    if prev_acc:
        accuracy = prev_acc
    else:
        accuracy = []

    while True:

        #reading each frame
        ret, frame = input_mov.read()
        frame_number = input_mov.get(1)

        #breaking when the video is over
        if not ret:
            break

        #finding all of the faces (+their encodings) in the current frame
        #for face_locations, I could either choose default "hog" or 'cnn'
        #"cnn" classifies better but it is CUDA accelerated and my computer
        #doesn't support that (i.e., after the last Catalina update)
        face_locs = face_recognition.face_locations(frame, model=model)
        face_encs = face_recognition.face_encodings(frame, face_locs)

        #repeating for each face we find
        for idx, (face_enc, face_loc) in enumerate(zip(face_encs, face_locs)):

            #comparing encodings (i.e., known faces with candidate encodings)
            #returns True if they match, and False if they don't
            results = face_recognition.compare_faces(faces, face_enc, tol)

            #additional measure which compares the candidate face with the known
            #faces and returns the euclidean distance between them (i.e., the␣
↪distance
            #defines how similar the faces are - smaller is better)
            distance = face_recognition.face_distance(faces, face_enc)

            #find the smallest distance face
            best = np.argmin(distance)

            #check if the smallest distance face (i.e., from known) passes the␣
↪tolerance
            if results[best]:
                match = names[best]
                prev = match
            else:
                #don't put a square if no face passes the tolerance (i.e., unknown␣
↪face or
```

```
                #not sure enough)
                continue

            if match==test[int(frame_number)]:
                accuracy.append(1)
            else:
                accuracy.append(0)

            #creating squares to highlight the faces
            top_left = (face_loc[3]-10, face_loc[0]-10)
            bottom_right = (face_loc[1]+10, face_loc[2]+10)

            #choosing white to be the color
            color = [255, 255, 255]

            cv2.rectangle(frame, top_left, bottom_right, color, 1)

            #defining the blocks for text
            top_left = (face_loc[3], face_loc[2]+10)
            bottom_right = (face_loc[1]+10, face_loc[2] + 32)

            cv2.rectangle(frame, top_left, bottom_right, color, cv2.FILLED)

            #defining the text
            cv2.putText(frame, match, (face_loc[3] + 5, face_loc[2] + 25),
                        cv2.FONT_ITALIC, 0.45, (0, 0, 0), 1)

        #writing the processed frames to the output file
        output_mov.write(frame)

    #processing and cleaning
    input_mov.release()
    output_mov.release()

    return accuracy
```

[4]:
```
#initialization
faces, names = encoding("faces")
```

[5]:
```
bs_test = []

#the correct classifications for the corresponding frames
#there are overall 479 frames and I labeled the frames manually
for i in range(41):
    bs_test.append("Suga")

for i in range(95):
    bs_test.append("V")

for i in range(54):
```

```
    bs_test.append("JK")

for i in range(50):
    bs_test.append("RM")

for i in range(93):
    bs_test.append("J-Hope")

for i in range(145):
    bs_test.append("Jin")
```

[6]:
```
#the process itself which returns the accuracy
accuracy = clf("bs.mov", "bs_face.mov", 0.5, 'cnn', test=bs_test)
```

[7]:
```
print("Testing accuracy (479 frames):", np.mean(accuracy))
```

```
Testing accuracy (479 frames): 0.8022598870056498
```

2 Appendix B: VGGFace

[8]:
```
#importing the needed libraries
import keras_vggface
from keras_vggface.vggface import VGGFace
from keras.layers import Input, Dense, Flatten, Dropout
from keras.models import Model
from keras.optimizers import Adam, SGD
```

[9]:
```
#loading the original VGGFace model without the top
vgg = VGGFace(include_top=False, input_shape=(224, 224, 3))

#setting the few layers as non-trainable because they usually capture
#the universal features that might be needed for our task too
for layer in vgg.layers:
    layer.trainable = False

last_layer = vgg.output

#building new layers for preventing overfitting
x = Flatten(name='flatten')(last_layer)
out = Dense(7, activation='softmax')(x)

model = Model(vgg.input, out)
```

[10]:
```
from keras.preprocessing.image import ImageDataGenerator

#preprocessing the images
batch_size = 4
nrow, ncol = 224, 224
```

```
epo = 5

#flowing all the images from the training directory
train_data_dir = './faces'
tr_datagen = ImageDataGenerator(rescale=1./255,
                                shear_range=0.2,
                                zoom_range=0.2)
train_generator = tr_datagen.flow_from_directory(
                        train_data_dir,
                        target_size=(nrow,ncol),
                        batch_size=batch_size,
                        class_mode='sparse')
```

```
Found 192 images belonging to 7 classes.
```

[11]:
```
#adding some arbitrary images to see how well the model
#can differentiate the members
test_data_dir = './test_faces'
te_datagen = ImageDataGenerator(rescale=1./255)
test_generator = te_datagen.flow_from_directory(
                        test_data_dir,
                        target_size=(nrow,ncol),
                        batch_size=batch_size,
                        class_mode='sparse')
```

```
Found 27 images belonging to 7 classes.
```

[12]:
```
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',␣
 ↪metrics='accuracy')
model.fit_generator(train_generator, steps_per_epoch=20,
    epochs=epo, validation_data=test_generator)
```

```
Epoch 1/5
20/20 [==============================] - 18s 864ms/step - loss: 1.9500 -
accuracy: 0.2004 - val_loss: 1.8473 - val_accuracy: 0.2222
Epoch 2/5
20/20 [==============================] - 18s 913ms/step - loss: 1.7843 -
accuracy: 0.3371 - val_loss: 1.6787 - val_accuracy: 0.4815
Epoch 3/5
20/20 [==============================] - 18s 890ms/step - loss: 1.4759 -
accuracy: 0.6600 - val_loss: 1.6598 - val_accuracy: 0.2963
Epoch 4/5
20/20 [==============================] - 17s 858ms/step - loss: 1.2229 -
accuracy: 0.7395 - val_loss: 1.5790 - val_accuracy: 0.4074
Epoch 5/5
20/20 [==============================] - 17s 864ms/step - loss: 1.1361 -
accuracy: 0.8624 - val_loss: 1.5512 - val_accuracy: 0.4815
```

[172]:
```
<tensorflow.python.keras.callbacks.History at 0x7f98f72c6d10>
```

[13]:
```
#setting the next layers as trainable
for layer in model.layers:
    layer.trainable = True

#performing a few more iterations of gradient descent with a small learning rate
model.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(lr=1e-7),␣
 ↪metrics='accuracy')
model.fit_generator(train_generator, steps_per_epoch=10, epochs=epo,␣
 ↪validation_data=test_generator)
```

```
Epoch 1/5
10/10 [==============================] - 29s 3s/step - loss: 0.9839 - accuracy:
0.9438 - val_loss: 1.5489 - val_accuracy: 0.4815
Epoch 2/5
10/10 [==============================] - 29s 3s/step - loss: 1.0220 - accuracy:
0.8710 - val_loss: 1.5466 - val_accuracy: 0.4815
Epoch 3/5
10/10 [==============================] - 28s 3s/step - loss: 1.0707 - accuracy:
0.8882 - val_loss: 1.5444 - val_accuracy: 0.4815
Epoch 4/5
10/10 [==============================] - 27s 3s/step - loss: 0.9482 - accuracy:
0.8175 - val_loss: 1.5422 - val_accuracy: 0.4815
Epoch 5/5
10/10 [==============================] - 27s 3s/step - loss: 1.2515 - accuracy:
0.6934 - val_loss: 1.5401 - val_accuracy: 0.4815
```

[13]:
```
<tensorflow.python.keras.callbacks.History at 0x7f98fa3bacd0>
```

[14]:
```
model.evaluate(test_generator)
```

```
7/7 [==============================] - 3s 466ms/step - loss: 1.5401 - accuracy:
0.4815
```

[14]:
```
[1.5401057004928589, 0.48148149251937866]
```

[15]:
```
import tensorflow as tf
import logging
tf.get_logger().setLevel(logging.ERROR)

def matching(frame, results, model, test, frame_number):
    """
    Function for reading the input video and initializing the output video.

    Input:
        frame: the current frame we need to process.
        results: the output from the face detection package.
        model: the classification model we are using.
        test: the test labels.
        frame_number: the current frame number.
```

```
    Output:
        frame: processed frame.
    """

    #going through all of the identified faces (i.e., locations)
    for result in results:
        x, y, w, h = result['box']
        #cropping out the faces so that it is easier to classify
        crop = frame[y:y+h, x:x+w]

        #preprocessing the frames so they would fit into the classifier
        output = cv2.cvtColor(crop, cv2.COLOR_BGR2RGB)
        output = cv2.resize(crop, (224, 224)).astype("float32")
        #### try with mean

        ####### try without [0]
        #predicting the class (i.e., gives probabilities for all seven
        #so we should extract the highest probability)
        prediction = model.predict(np.expand_dims(output, axis=0))[0]
        y_pred = prediction.argmax(axis=-1)

        #finding the name of the match
        match = list(labels.keys())[list(labels.values()).index(y_pred)]

        #drawing the rectangles
        cv2.rectangle(frame, (x-10, y-10), (x+w+10, y+h+10),
                  (255, 255, 255), 1)
        cv2.rectangle(frame, (x-10, y+h+10), (x+w+10, y+h+30),
                      (255, 255, 255), cv2.FILLED)

        cv2.putText(frame, match, (x + 5, y+h+25),
                cv2.FONT_ITALIC, 0.45, (0, 0, 0), 1)

        #testing
        if match==test[int(frame_number)]:
            accuracy.append(1)
        else:
            accuracy.append(0)

    return frame, accuracy
```

[16]:
```
from mtcnn import MTCNN
def vgg_clf(inp, out, model, test=None, prev_acc=None):
    """
    Function for reading the input video and initializing the output video.

    Input:
        inp: path to the input video that we want to process.
        tol: path to the input video that we want to process.
        model: the NN model to use for classification.
```

```
    Output:
        accuracy: the list with 0/1 depending on whether the
        classification is correct.
    """

    input_mov, output_mov = in_out(inp, out)
    mean = np.array([123.68, 116.779, 103.939][::1], dtype="float32")

    #seeing whether we should keep track of some previous accuracy
    if prev_acc:
        accuracy = prev_acc
    else:
        accuracy = []

    while True:

        #reading each frame
        ret, frame = input_mov.read()
        frame_number = input_mov.get(1)

        #breaking when the video is over
        if not ret:
            break

        #detecting faces and classifying the people
        detector = MTCNN()
        results = detector.detect_faces(frame)
        new_frame, accuracy = matching(frame, results, model, test, frame_number)

        #writing the processed frames to the output file
        output_mov.write(new_frame)

    #processing and cleaning
    input_mov.release()
    output_mov.release()

    return accuracy
```

[18]:
```
labels = train_generator.class_indices

#the process itself which returns the accuracy
accuracy = vgg_clf('bs.mov', "bs_vgg.mov", model, test=bs_test)
```

[19]:
```
print("Testing accuracy (479 frames):", np.mean(accuracy))
```

```
Testing accuracy (479 frames): 0.65
```