HO CHI MINH UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MACHINE LEARNING

REPORT
FACE IDENTITY RECOGNITION

LECTURER:	Nguyen Duc Dung, PhD.
CLASS:	CC01
STUDENT:	Pham Tan Phuoc - 1952406

HO CHI MINH CITY, 12/2021

Contents

Chapter 1

Introduction

1.1 What is face identity recognition?

Before diving into the details, we first need to know the meaning of this concept. To put it simply, it is a technology that identifies and verifies an individual from a digital image and video. It captures, examines, and then compares a person's facial details.

While the concept of facial recognition is not new, technological improvement has led to a massive proliferation of this technology. Today, you can see its usage everywhere – in passport offices to detect fraud in passports and travel visas; in airports to screen passengers; in police stations to scan the criminal data; in ATMs/banks to ensure the security of users; in organizations to keep track on the attendance of the employees, and more.

Figure 1.1: Face identity recognition.

In addition to this, some web applications, social networking websites, and smartphones also use this technology to prevent data misuse. Probably, your Smartphone also has a face recognition feature to unlock it. With these information, it is ready for getting on the next section.

1.2 How to apply Machine Learning?

In Machine Learning field, there are several isolated problems that have only one step such as estimating the price of a house, generating new data based on existing data and telling if an image contains a certain object. However, face identity recognition is on a different level, it requires a series of several related problems:

1. First, look at a picture and find all the faces in it.
2. Second, focus on each face and be able to understand that even if a face is turned in a weird direction or in bad lighting, it is still the same person.
3. Third, be able to pick out unique features of the face that you can use to tell it apart from other people— like how big the eyes are, how long the face is, etc.
4. Finally, compare the unique features of that face to all the people you already know to determine the person's name.

As a human, your brain is wired to do all of this automatically and instantly. In fact, humans are too good at recognizing faces and end up seeing faces in everyday objects:

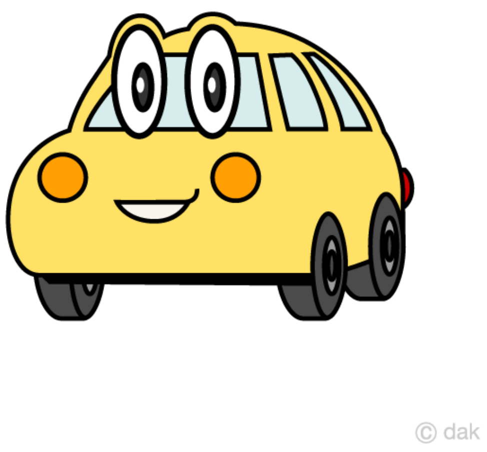


Figure 1.2: Human ability to recognize faces.

However, computers are not capable of this kind of high-level generalization (at least not yet...), so we have to teach them how to do each step in this process separately.

We need to build a pipeline where we solve each step of face recognition separately and pass the result of the current step to the next step. In other words, we will chain together several machine learning algorithms.

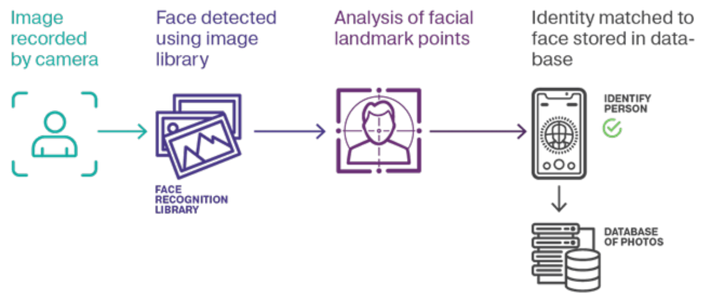


Image recorded by camera
Face detected using image library
Analysis of facial landmark points
Identity matched to face stored in data-base
FACE RECOGNITION LIBRARY
IDENTIFY PERSON
DATABASE OF PHOTOS

Figure 1.3: Face identity recognition process.

Chapter 2

Face Identity Recognition - Step by Step

In this section, I am going to split this problem into steps. For each step, I'll explain and apply a suitable machine learning algorithm for our subproblem. Since the target of this assignment is to understand the idea of the method for performing face identity recognition, I'm not going to explain every single algorithm completely, the main content is just the main idea behind each algorithm and how I use Python to implement.

2.1 Face detection

Before doing anything, obviously we need to know where faces located in an image. Therefore, the first step in our pipeline is face detection. Basically, face detection is a problem in computer vision of locating and localizing one or more faces in a photograph. Locating a face in a photograph refers to finding the coordinate of the face in the image, whereas localization refers to demarcating the extent of the face, often via a bounding box around the face.

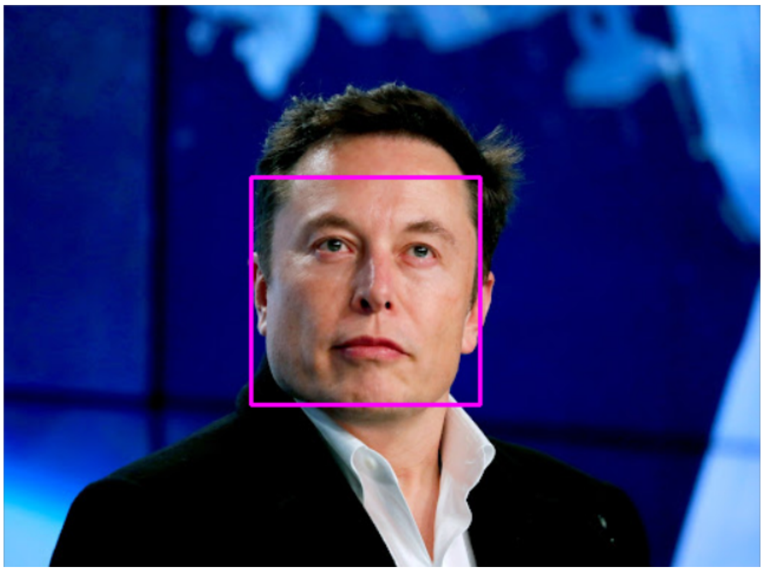


Figure 2.1: Face detection.

These days, there have been a lot of papers and tutorials related to face detection. However, to make it simple, I will not go into details about it anymore, instead, I am going to use the face_recognition library to complete this task. Besides, you can also use other libraries such as OpenCV, which is a library that provide Classifier Cascade face detection algorithm to do this. Now, the below lines of code will show you how the face_recognition library can perform face detection:

```python
import cv2
import face_recognition

image = face_recognition.load_image_file('Path_to_image.jpg')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
face_loc = face_recognition.face_locations(image)
```

The above lines of code will first load an image from a given path. After loading the image, we need to convert it back to RGB ordering with the function cvtColor from OpenCV library since OpenCV always reads images in BGR ordering but the library understands it as RGB. Then, when the face_locations() function is executed, it will return an array of bounding boxes of human faces in the image. The default model of this function is “HOG”, which stands for Histogram of Oriented Gradients.

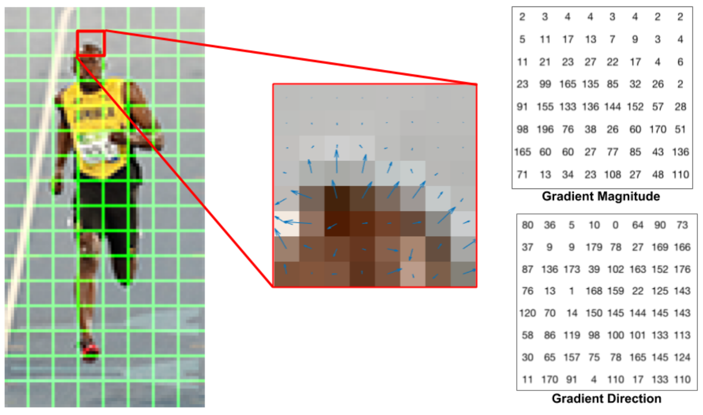
How HOG works

The Histogram of Oriented Gradients (HOG) is an efficient way to extract features out of the pixel colors for building an object recognition classifier, or in our case - face recognition classifier, by using the knowledge of image gradient vectors.

HOG includes four main steps:

1. Preprocess the image, including resizing and color normalization.
2. Compute the gradient vector of every pixel, as well as its magnitude and direction.
3. Divide the image into many 8x8 pixel cells. In each cell, the magnitude values of these 64 cells are binned and cumulatively added into 9 buckets of unsigned direction (no sign, so 0-180 degree rather than 0-360 degree; this is a practical choice based on empirical experiments).
4. Then we slide a 2x2 cells (thus 16x16 pixels) block across the image. In each block region, 4 histograms of 4 cells are concatenated into one-dimensional vector of 36 values and then normalized to have an unit weight. The final HOG feature vector is the concatenation of all the block vectors. It can be fed into a classifier like SVM for learning object recognition tasks.

Let's look at one 8×8 patch in an image and see how the gradients look.



2	3	4	4	3	4	2	2
5	11	17	13	7	9	3	4
11	21	23	27	22	17	4	6
23	99	165	135	85	32	26	2
91	155	133	136	144	152	57	28
98	196	76	38	26	60	170	51
165	60	60	27	77	85	43	136
71	13	34	23	108	27	48	110

Gradient Magnitude

80	36	5	10	0	64	90	73
37	9	9	179	78	27	169	166
87	136	173	39	102	163	152	176
76	13	1	168	159	22	125	143
120	70	14	150	145	144	145	143
58	86	119	98	100	101	133	113
30	65	157	75	78	165	145	124
11	170	91	4	110	17	133	110

Gradient Direction

Figure 2.2: Histogram of Oriented Gradients.

Now let's get back to our face detection problem, to make it clearer, I will perform the above lines of code on a sample image:

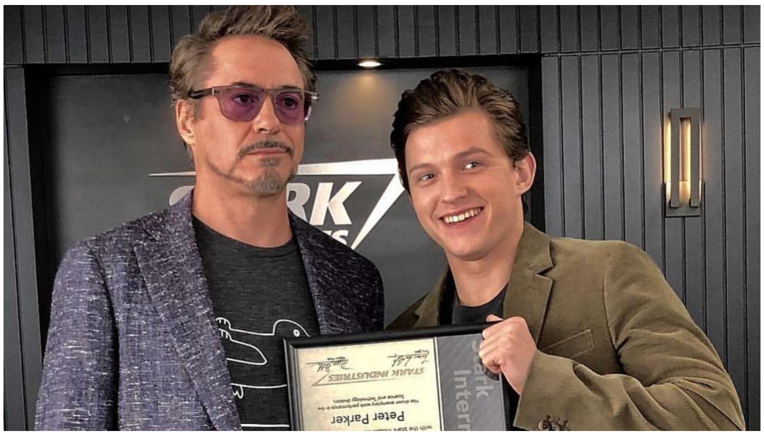


Figure 2.3: Sample image.

With this image, the above code will return an array as below:

```
[(192, 860, 415, 637), (98, 531, 284, 345)]
```

Those are the locations of every face occuring in the image. And for now, we can see that there are two locations which correspond to two faces in the image. With these locations, we are able to draw some rectangles on these faces to verify that our code works well. The following lines of code will help us to do that:

```python
for face in face_loc:
    y1, x2, y2, x1 = face
    cv2.rectangle(image, (x1, y1), (x2, y2), (0, 255, 0), 2)
cv2.imshow('Tom and Stark', image)
cv2.waitKey(0)
```

The result is as below:

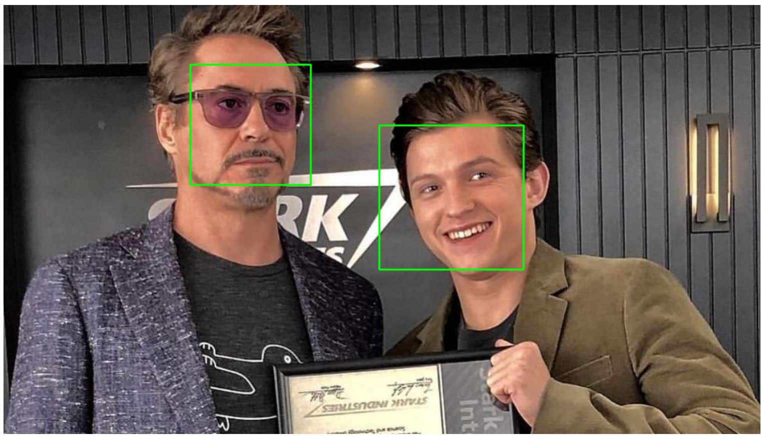


Figure 2.4: Result of face detection.

The code seems to be very good, now we are ready for the next step, which is encoding faces.

2.2 Encoding faces

Now we have detected all faces in an image successfully, we are ready to perform the next step, which will actually tell faces apart. The simplest approach to face recognition is to directly compare the unknown face we have detected in step 1 with all labeled pictures in the database. However, this method is not really practical, since a site like Facebook with billions of users and a trillion photos can't possibly loop through every previous-tagged face to compare it to every newly uploaded picture. That would take way too long.

Therefore, it comes up with encoding faces, which is a way to extract a few basic measurements from each face. Then we could measure our unknown face the same way and find the known face with the closest measurements. For example, we might measure the size of each ear, the spacing between the eyes, the length of the nose, etc. So how we can collect those measurements from our known face database? Luckily, the face_recognition library also supports us with a function named face_encodings, which helps us to get the 128-dimension face encoding for each face in the image.

```
encode_Face = face_recognition.face_encodings(image)
print(encode_Face)
```

The "image" in the code is the sample image of **Figure 2.2**. The result is as below:

```
[array([-1.54029146e-01,  8.47455487e-02, 7.33846724e-02, ..., 1.07917495e-01]),
 array([-0.06038005,  0.05374614,  0.09215817, ..., 0.07926586])]
```

The above result gives us 128 measurements for each face in the image. And with these measurements, we will be able to use a machine learning classification algorithm to perform the last step.

2.3 Finding the person's name from the encoding

In this step, what we have to do is find the person in our database of known people who has the closest measurements to our test image. There are many machine learning classification algorithms that we can use for this task. However, I will use a simple linear SVM classifier to do this.

Basically, all we need to do is train the SVM classifier that can take in the measurements from a new test image and tells which known person is the closest match. Running this classifier takes milliseconds. The result of the classifier is the name of the person.

What is SVM?

"Support Vector Machine" (SVM) is a supervised machine learning algorithm that can be used for both classification or regression challenges. However, it is mostly used in classification problems. In the SVM algorithm, we plot each data item as a point in n-dimensional space (where n is a number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiates the two classes very well.

So how to find the best hyper-plane? One reasonable choice as the best hyperplane is the one that represents the largest separation or margin between the two classes. Simply, we choose the hyperplane whose distance from it to the nearest data point on each side is maximized. If such a hyperplane exists it is known as the maximum-margin hyperplane/hard margin. Look at the below figure:

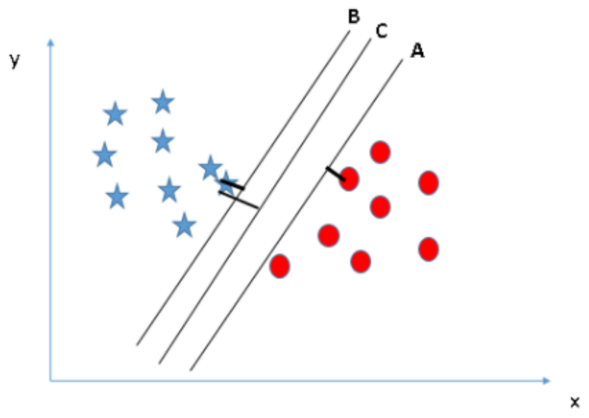


Figure 2.5: SVM algorithm.

Above, you can see that the margin for hyper-plane C is high as compared to both A and B. Hence, we name the right hyper-plane as C. The reason for selecting the hyper-plane with higher margin is robustness. If we select a hyper-plane having low margin then there is high chance of miss-classification. So, let's imagine if we apply this for our face identity recognition problem, consider that we have A and B as the people we have to recognize, the encodings of A will be the blue star in the above figure and B will be the red circle. What SVM classifier will do is that it will try to find the best hyperplane which will seperate these measurements/encodings into two groups. With this, anytime new encodings from a new image get into our model, the SVM classifier will be able to know which group it belongs to easily.

Now, we will move to the coding part. Let's see the code first:

```python
# The training data would be all the face encodings from all the known images
# The labels are their names
encodings = []
names = []

# Training directory
train_dir = os.listdir('SVM Classifier/train_dir/')

# Loop through each person in the training directory
for person in train_dir:
    pix = os.listdir("SVM Classifier/train_dir/" + person)

    # Loop through each training image for the current person
    for person_img in pix:
        # Get the face encodings for the face in each image file
        face = face_recognition.load_image_file("SVM Classifier/train_dir/" +
    person + "/" + person_img)
        face_bounding_boxes = face_recognition.face_locations(face)

        #If training image contains exactly one face
        if len(face_bounding_boxes) == 1:
            face_enc = face_recognition.face_encodings(face)[0]
            # Add face encoding for current image with corresponding label
            # (name) to the training data
            encodings.append(face_enc)
            names.append(person)
        else:
            print(person + "/" + person_img + " was skipped and can't be used for
    training")

# Create and train the SVC classifier
clf = svm.SVC(gamma='scale')
clf.fit(encodings,names)
```

These lines of code will first create two empty lists, which are encodings and names, we are going to use these two lists for training our model later. To append these empty lists with encodings and names, we will loop over our training directory. This is the structure of our directory:

```
Structure:
        <test_image>.jpg
        <train_dir>/
            <person_1>/
                <person_1_face-1>.jpg
                <person_1_face-2>.jpg
                .
                .
                <person_1_face-n>.jpg
           <person_2>/
                <person_2_face-1>.jpg
                <person_2_face-2>.jpg
                .
                .
                <person_2_face-n>.jpg
            .
            .
            <person_n>/
                <person_n_face-1>.jpg
                <person_n_face-2>.jpg
                .
                .
                <person_n_face-n>.jpg
```

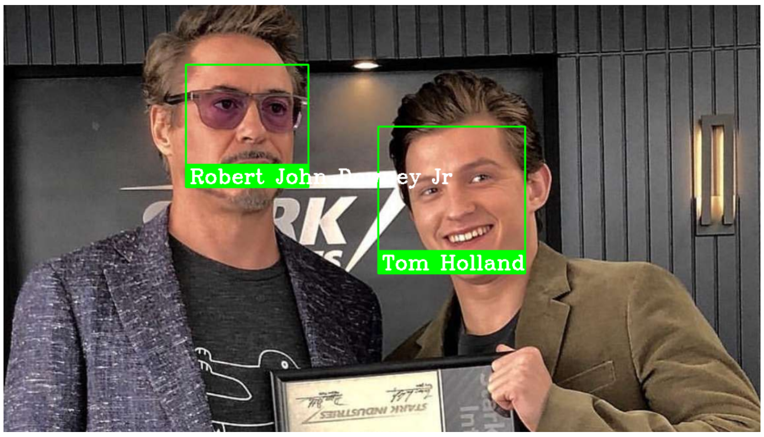
With this structure, each of our loops will be for a person in our dataset. And inside the outer loop, there are also an inner loop to get through every training image of a person. At the end, the encodings and names list will be filled with all encodings and names from all images in our training directory. Using these two, we are ready for the training step.

```python
# Create and train the SVC classifier
clf = svm.SVC(gamma='scale')
clf.fit(encodings,names)
```

Since Scikit-learn has already provided us a method to create and train the SVM classifier, we will be able to do this step by just two lines of code. Now we will perform the following lines of code to test our result:

```python
# Load the test image with unknown faces into a numpy array
test_image = face_recognition.load_image_file('SVM Classifier/test_image.jpg')

# Find all the faces in the test image using the default HOG-based model
test_image = cv2.cvtColor(test_image, cv2.COLOR_BGR2RGB)
face_locations = face_recognition.face_locations(test_image)
face_encodings = face_recognition.face_encodings(test_image)

for encodeFace, faceLoc in zip(face_encodings, face_locations):
        name = clf.predict([encodeFace])
        y1, x2, y2, x1 = faceLoc
        cv2.rectangle(test_image, (x1, y1), (x2, y2), (0, 255, 0), 2)
        cv2. rectangle(test_image, (x1, y2 - 35), (x2,y2),(0, 255, 0), cv2.FILLED)
        cv2.putText(test_image, *name, (x1 + 6, y2 - 6), cv2.FONT_HERSHEY_COMPLEX
    ,1,(255,255,255),2)

cv2.imshow('Test', test_image)
cv2.waitKey(0)
```

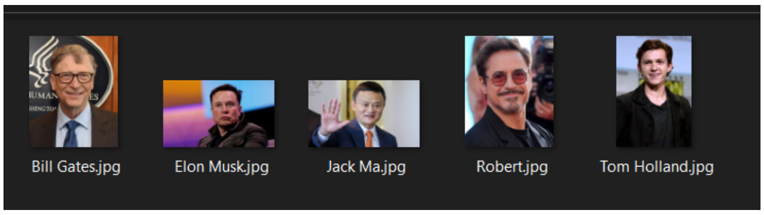
The result:



2.4 Application

That's all for the implementation. Now I want to show you a simple application that this model can be applied in practice. This application is to check attendance, which may useful for companies and schools. Let's see how it works.

Let's say we have Bill Gates, Jack Ma, Elon Musk, Robert and Tom Holland as the people we need to check attendance. We first need to upload some sample images of them to train the model.



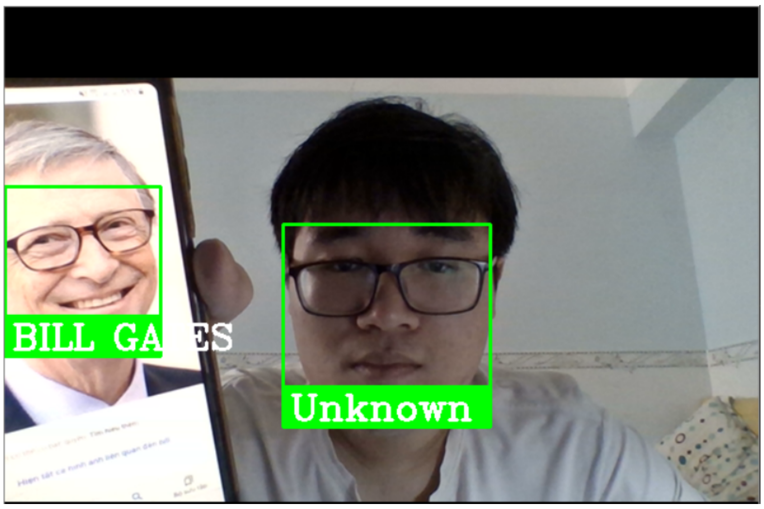
And next, by adding the following lines of code, we can connect to the webcam for checking attendance of everyone whose faces had been get through it.

```python
# Use webcam to get test_image
cap = cv2.VideoCapture(0)

while True:
    success, img = cap.read()
    # Resize smaller to speed up performance
    imgS = cv2.resize(img, (0,0), None, 0.25, 0.25)
    imgS = cv2.cvtColor(imgS, cv2.COLOR_BGR2RGB)

    facesCurFrame = face_recognition.face_locations(imgS) # May have many faces
    encodesCurFrame = face_recognition.face_encodings(imgS, facesCurFrame)

    for encodeFace, faceLoc in zip(encodesCurFrame, facesCurFrame):
        name = clf.predict([encodeFace])
        y1, x2, y2, x1 = faceLoc
        y1, x2, y2, x1 = y1*4, x2*4, y2*4, x1*4
        cv2.rectangle(img, (x1, y1), (x2, y2), (0, 255, 0), 2)
        cv2. rectangle(img, (x1, y2 - 35), (x2,y2),(0, 255, 0), cv2.FILLED)
        cv2.putText(img, *name, (x1 + 6, y2 - 6), cv2.FONT_HERSHEY_COMPLEX
    ,1,(255,255,255),2)

    cv2.imshow('Webcam', img)
    cv2.waitKey(1)
```

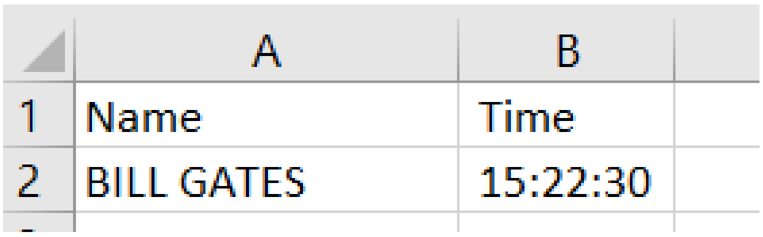
Now I will add a small function to the code for storing the data in a .csv file:

```python
def markAttendance(name):
    with open('Attendance.csv', 'r+') as f:
        myDataList = f.readlines()
        nameList = []
        for line in myDataList:
            entry = line.split(',')
            nameList.append(entry[0])
        if name not in nameList:
            now = datetime.now()
            dtString = now.strftime('%H:%M:%S')
            f.writelines(f'\n{name}, {dtString}')
```

With these, we now have a simple application that can help us to check attendance, as below:



Using the above moment of webcam, the .csv file will store the data as follow:



	A	B
1	Name	Time
2	BILL GATES	15:22:30

Of course, this is just a small and basic application of this problem, using your ideas, you will able to develop lots of other applications which are more interesting and useful.

Chapter 3

Conclusion

Face recognition technology has come a long way in the last twenty years. Today, machines are able to automatically verify identity information for secure transactions, for surveillance and security tasks, and for access control to buildings etc. These applications usually work in controlled environments and recognition algorithms can take advantage of the environmental constraints to obtain high recognition accuracy. Hopefully, next generation face recognition systems will become a widespread application, which are going to be helpful assistants for humanity.

That's all about what I want to perform for this topic. Of course, my model still has some limitations with the accuracy and so on. However, I hope that this report can cover the most basic parts of this face identity recognition problem.

At last, with gratitude, many thanks to PhD. Nguyen Duc Dung for giving us a chance to learn and explore this topic, which is very interesting and really valuable in our career path.

Github link: link.

Bibliography

[1] GeeksforGeeks. *Support Vector Machine Algorithm*. Available at: https://www.geeksforgeeks.org/support-vector-machine-algorithm/

[2] Sunil Ray. *Understanding Support Vector Machine(SVM) algorithm from examples (along with code)*. Available at: https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

[3] Dr.Adrian Rosebrock. *Deep Learning for Computer Vision.*

[4] Adam Geitgey. *Modern Face Recognition with Deep Learning*. Available at: https://medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-face-recognition-with-deep-learning-c3cffc121d78

[5] Object Detection for Dummies Part 1: Gradient Vector, HOG, and SS. Available at: https://lilianweng.github.io/lil-log/2017/10/29/object-recognition-for-dummies-part-1.html#histogram-of-oriented-gradients-hog

[6] Jason Brownlee. *How to Perform Face Detection with Deep Learning*. Available at: https://machinelearningmastery.com/how-to-perform-face-detection-with-classical-and-deep-learning-methods-in-python-with-keras/

[7] Satya Mallick. *Histogram of Oriented Gradients explained using OpenCV*. Available at: https://learnopencv.com/histogram-of-oriented-gradients/